Unleashing the Power of Asynchronous HTTP with Ruby

Samuel Williams
https://ruby.social/@ioquatix

(Breathing, Pacing, Lanyard, Water, Eye Contact, Power, Clock)

Unleashing the Power of Asynchronous HTTP with Ruby

Samuel Williams
https://ruby.social/@ioquatix

Hello everyone! My name is Samuel Williams and today we will be Unleashing the power of Asynchronous HTTP with Ruby!

https://github.com/ioquatix/rubykaigi-2023

プレゼンテーションのソースコード

All the source code shown in this presentation is available online, so you can try it out for yourself. Also note that the example code is for demonstration purpose only and is not a full or complete implementation.

What is HTTP and why is it important?

HTTP とは何ですか?なぜ重要ですか?

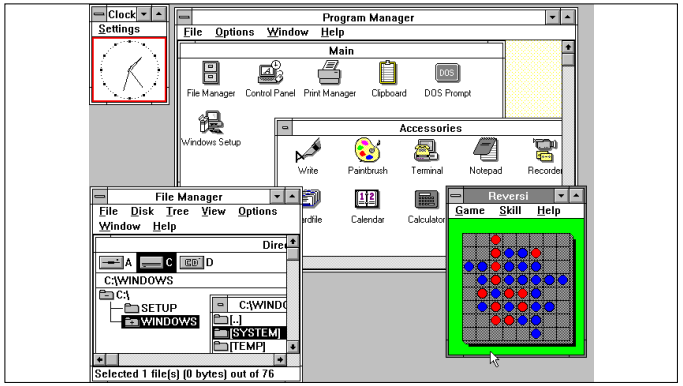
So, what is HTTP and why is it important?

1990
33 years ago

1990年
33年前

To answer this question, let’s go back in time to 1990, 33 years ago.

The Hubble Space Telescope was launched into orbit by NASA, providing unprecedented views of the universe.



Windows 3.0 was just released, which would go on to shape the future of the personal computer.

The first version of Adobe Photoshop revolutionised the creation of digital art work.

The birth of the internet.

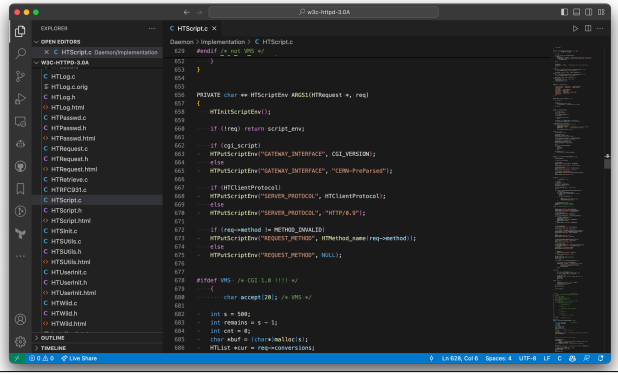
インターネットの誕生。

And Sir Tim Berners-Lee created the first web server, web browser, and web site. It was the birth of the internet.

The first web server was called **CERN httpd** and was used to serve the first websites.

最初の Web サーバーは CERN httpd と呼ばれ、最初の Web サイトを提供するために使用されました。

He created the first web server, called CERN httpd. It was used to serve the first websites.



The last version of the source code, which was released in 1996, is available online.

The first website was hosted on a **NeXT** computer at **CERN**.

最初の Web サイトは CERN の NeXT コンピューターでホストされていました。

He also created the first website, which was hosted on a NeXT computer at CERN.

```
<HEADER>
<TITLE>The World Wide Web project</TITLE>
<NEXTID N="55">
</HEADER>
<BODY>
<H1>World Wide Web</H1>The WorldWideWeb (W3) is a wide-area<A
NAME=0 HREF="WhatIs.html">
hypermedia</A> information retrieval
initiative aiming to give universal
access to a large universe of documents.<P>
Everything there is online about
W3 is linked directly or indirectly
to this document, including an <A
NAME=24 HREF="Summary.html">executive
summary</A> of the project, <A
NAME=29 HREF="Administration/Mailing/Overview.html">Mailing lists</A>
, <A
NAME=30 HREF="Policy.html">Policy</A> , November's  <A
NAME=34 HREF="News/9211.html">W3  news</A> ,
```

It provided information about the World Wide Web project itself, using Hyper-Text Markup Language.

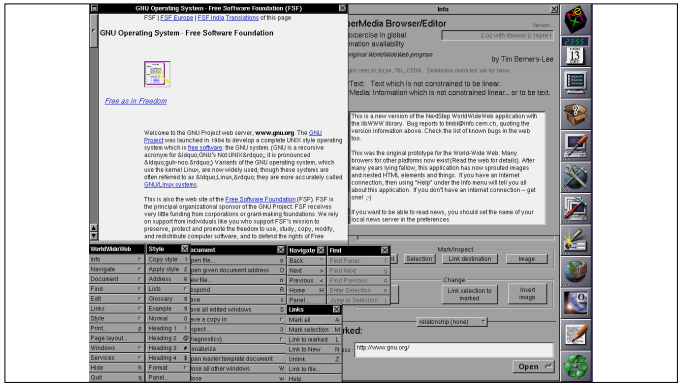
```
<DT><A
NAME=14 HREF="People.html">People</A>
<DD> A list of some people involved
in the project.
<DT><A
NAME=15 HREF="History.html">History</A>
<DD> A summary of the history
of the project.
<DT><A
NAME=37 HREF="Helping.html">How can I help</A> ?
<DD> If you would like
to support the web..
<DT><A
NAME=48 HREF="../README.html">Getting code</A>
<DD> Getting the code by<A
NAME=49 HREF="LineMode/Defaults/Distribution.html">
anonymous FTP</A> , etc.</A>
</DL>
</BODY>
```

HTML still looks similar today, as it did 33 years ago.

The first web browser was called **WorldWideWeb** which also included an editor.

最初の Web ブラウザは WorldWideWeb と呼ばれ、エディタも含まれていました。

Finally, he created the first web browser called WorldWideWeb, which also included an editor.



By today's standards, it looks a bit cluttered, but it was a revolutionary new approach for generating and sharing content.

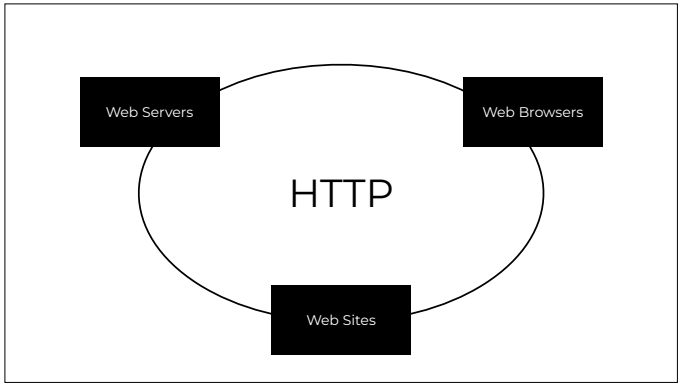
What was the most important technology?

最も重要な技術は何でしたか?

So out of these technologies, what was the most important?

Hyper-Text Transfer Protocol

The Hyper-Text Transfer Protocol, or HTTP



…is the foundation on which web browsers, web servers and web sites are built. So let's take a look at the evolution of HTTP to understand how it works, and why it's important.

HTTP/0.9

(2:30) The very first release was named HTTP/0.9.

Developed during 1990 as a simple protocol for transferring hypertext documents.

1990 年にハイパーテキスト ドキュメントを転送するための単純なプロトコルとして開発されました。

It was developed during 1990 as a simple protocol for transferring hypertext documents. The official specification, which is about 700 words long, was released on January 1st, 1991.

Only supported GET method and had no support for headers or other HTTP methods.

GET メソッドのみがサポートされ、ヘッダーや他の HTTP メソッドはサポートされませんでした。

It only supported a single GET method and had no support for request or response headers.

Only supported plain text responses, no support for multimedia or other types of content.

サポートされるのはプレーン テキスト応答のみで、マルチメディアやその他の種類のコンテンツはサポートされません。

Because of that, it could not support other types of content like images, audio or video.

The protocol was used primarily for academic purposes and was not widely adopted by industry.

このプロトコルは主に学術目的で使用され、産業界では広く採用されませんでした。

As a prototype, it was used primarily for academic purposes and experimentation.

HTTP/0.9 Client

Let's take a look at the implementation of an HTTP/0.9 client.

```ruby
#!/usr/bin/env ruby

require 'async'
require 'async/io'
require 'async/io/stream'

Sync do
  endpoint = Async::IO::Endpoint.tcp('localhost', 8009)

  ARGV.each do |path|
    endpoint.connect do |connection|
      stream = Async::IO::Stream.new(connection)
      stream.write("GET #{path}\r\n")
      stream.flush
      puts stream.read
    end
  end
end
```

As you can see, it's short. This is the whole implementation in Ruby.

```ruby
#!/usr/bin/env ruby

require 'async'
require 'async/io'
require 'async/io/stream'

Sync do
  endpoint = Async::IO::Endpoint.tcp('localhost', 8009)

  ARGV.each do |path|
    endpoint.connect do |connection|
      stream = Async::IO::Stream.new(connection)
      stream.write("GET #{path}\r\n")
      stream.flush
      puts stream.read
    end
  end
end
```

We are using Async for the networking.

```ruby
#!/usr/bin/env ruby

require 'async'
require 'async/io'
require 'async/io/stream'

Sync do
  endpoint = Async::IO::Endpoint.tcp('localhost', 8009)

  ARGV.each do |path|
    endpoint.connect do |connection|
      stream = Async::IO::Stream.new(connection)
      stream.write("GET #{path}\r\n")
      stream.flush
      puts stream.read
    end
  end
end
```

We use a Sync block to create an event loop.

```ruby
#!/usr/bin/env ruby

require 'async'
require 'async/io'
require 'async/io/stream'

Sync do
  endpoint = Async::IO::Endpoint.tcp('localhost', 8009)

  ARGV.each do |path|
    endpoint.connect do |connection|
      stream = Async::IO::Stream.new(connection)
      stream.write("GET #{path}\r\n")
      stream.flush
      puts stream.read
    end
  end
end
```

We specify the endpoint for our client to connect to.

```ruby
#!/usr/bin/env ruby

require 'async'
require 'async/io'
require 'async/io/stream'

Sync do
  endpoint = Async::IO::Endpoint.tcp('localhost', 8009)

  ARGV.each do |path|
    endpoint.connect do |connection|
      stream = Async::IO::Stream.new(connection)
      stream.write("GET #{path}\r\n")
      stream.flush
      puts stream.read
    end
  end
end
```

Then for each argument on the command line…

```ruby
#!/usr/bin/env ruby

require 'async'
require 'async/io'
require 'async/io/stream'

Sync do
  endpoint = Async::IO::Endpoint.tcp('localhost', 8009)

  ARGV.each do |path|
    endpoint.connect do |connection|
      stream = Async::IO::Stream.new(connection)
      stream.write("GET #{path}\r\n")
      stream.flush
      puts stream.read
    end
  end
end
```

We connect to the server…

```ruby
#!/usr/bin/env ruby

require 'async'
require 'async/io'
require 'async/io/stream'

Sync do
  endpoint = Async::IO::Endpoint.tcp('localhost', 8009)

  ARGV.each do |path|
    endpoint.connect do |connection|
      stream = Async::IO::Stream.new(connection)
      stream.write("GET #{path}\r\n")
      stream.flush
      puts stream.read
    end
  end
end
```

We create a buffered stream…

```ruby
#!/usr/bin/env ruby

require 'async'
require 'async/io'
require 'async/io/stream'

Sync do
  endpoint = Async::IO::Endpoint.tcp('localhost', 8009)

  ARGV.each do |path|
    endpoint.connect do |connection|
      stream = Async::IO::Stream.new(connection)
      stream.write("GET #{path}\r\n")
      stream.flush
      puts stream.read
    end
  end
end
```

We write the request to the stream which includes the GET method and path…

```ruby
#!/usr/bin/env ruby

require 'async'
require 'async/io'
require 'async/io/stream'

Sync do
  endpoint = Async::IO::Endpoint.tcp('localhost', 8009)

  ARGV.each do |path|
    endpoint.connect do |connection|
      stream = Async::IO::Stream.new(connection)
      stream.write("GET #{path}\r\n")
      stream.flush
      puts stream.read
    end
  end
end
```

We flush the output…

```ruby
#!/usr/bin/env ruby

require 'async'
require 'async/io'
require 'async/io/stream'

Sync do
  endpoint = Async::IO::Endpoint.tcp('localhost', 8009)

  ARGV.each do |path|
    endpoint.connect do |connection|
      stream = Async::IO::Stream.new(connection)
      stream.write("GET #{path}\r\n")
      stream.flush
      puts stream.read
    end
  end
end
```

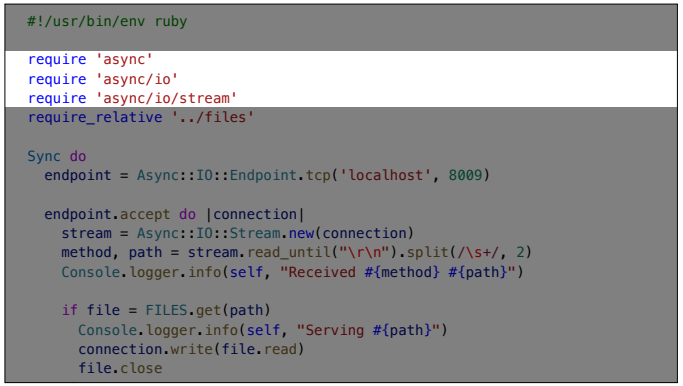
Then we wait for the response body and read it all.

HTTP/0.9 Server

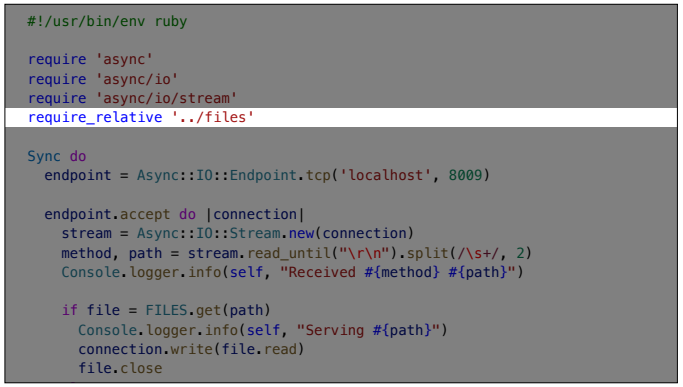
Now let’s look at an HTTP/0.9 server.

```ruby
#!/usr/bin/env ruby

require 'async'
require 'async/io'
require 'async/io/stream'
require_relative '../files'

Sync do
  endpoint = Async::IO::Endpoint.tcp('localhost', 8009)

  endpoint.accept do |connection|
    stream = Async::IO::Stream.new(connection)
    method, path = stream.read_until("\r\n").split(/\s+/, 2)
    Console.logger.info(self, "Received #{method} #{path}")

    if file = FILES.get(path)
      Console.logger.info(self, "Serving #{path}")
      connection.write(file.read)
      file.close
```

It's a little more complex than the client.



Again, we are implementing it using Async.



We also have our application, a file server.

```ruby
#!/usr/bin/env ruby

require 'async'
require 'async/io'
require 'async/io/stream'
require_relative '../files'

Sync do
  endpoint = Async::IO::Endpoint.tcp('localhost', 8009)

  endpoint.accept do |connection|
    stream = Async::IO::Stream.new(connection)
    method, path = stream.read_until("\r\n").split(/\s+/, 2)
    Console.logger.info(self, "Received #{method} #{path}")

    if file = FILES.get(path)
      Console.logger.info(self, "Serving #{path}")
      connection.write(file.read)
      file.close
```

We create a top level event loop.

```ruby
#!/usr/bin/env ruby

require 'async'
require 'async/io'
require 'async/io/stream'
require_relative '../files'

Sync do
  endpoint = Async::IO::Endpoint.tcp('localhost', 8009)

  endpoint.accept do |connection|
    stream = Async::IO::Stream.new(connection)
    method, path = stream.read_until("\r\n").split(/\s+/, 2)
    Console.logger.info(self, "Received #{method} #{path}")

    if file = FILES.get(path)
      Console.logger.info(self, "Serving #{path}")
      connection.write(file.read)
      file.close
```

We specify the endpoint we want to bind to.

```ruby
#!/usr/bin/env ruby

require 'async'
require 'async/io'
require 'async/io/stream'
require_relative '../files'

Sync do
  endpoint = Async::IO::Endpoint.tcp('localhost', 8009)

  endpoint.accept do |connection|
    stream = Async::IO::Stream.new(connection)
    method, path = stream.read_until("\r\n").split(/\s+/, 2)
    Console.logger.info(self, "Received #{method} #{path}")

    if file = FILES.get(path)
      Console.logger.info(self, "Serving #{path}")
      connection.write(file.read)
      file.close
```

We accept an incoming connection…

```ruby
Sync do
  endpoint = Async::IO::Endpoint.tcp('localhost', 8009)

  endpoint.accept do |connection|
    stream = Async::IO::Stream.new(connection)
    method, path = stream.read_until("\r\n").split(/\s+/, 2)
    Console.logger.info(self, "Received #{method} #{path}")

    if file = FILES.get(path)
      Console.logger.info(self, "Serving #{path}")
      connection.write(file.read)
      file.close
    else
      Console.logger.warn(self, "Could not find #{path}")
    end

  rescue => error
    Console.logger.error(self, error)
```

We create a buffered stream for that connection…

```ruby
Sync do
  endpoint = Async::IO::Endpoint.tcp('localhost', 8009)

  endpoint.accept do |connection|
    stream = Async::IO::Stream.new(connection)
    method, path = stream.read_until("\r\n").split(/\s+/, 2)
    Console.logger.info(self, "Received #{method} #{path}")

    if file = FILES.get(path)
      Console.logger.info(self, "Serving #{path}")
      connection.write(file.read)
      file.close
    else
      Console.logger.warn(self, "Could not find #{path}")
    end

  rescue => error
    Console.logger.error(self, error)
```

We read the request line, which includes the method and the path.

```ruby
Sync do
  endpoint = Async::IO::Endpoint.tcp('localhost', 8009)

  endpoint.accept do |connection|
    stream = Async::IO::Stream.new(connection)
    method, path = stream.read_until("\r\n").split(/\s+/, 2)
    Console.logger.info(self, "Received #{method} #{path}")

    if file = FILES.get(path)
      Console.logger.info(self, "Serving #{path}")
      connection.write(file.read)
      file.close
    else
      Console.logger.warn(self, "Could not find #{path}")
    end

  rescue => error
    Console.logger.error(self, error)
```

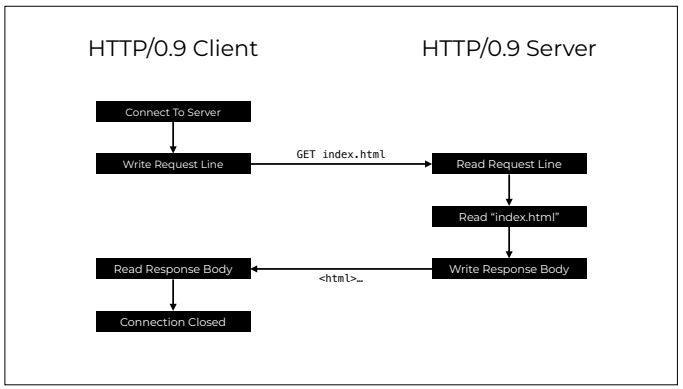
Then we ask the file system for the file at that path.

```ruby
Sync do
  endpoint = Async::IO::Endpoint.tcp('localhost', 8009)

  endpoint.accept do |connection|
    stream = Async::IO::Stream.new(connection)
    method, path = stream.read_until("\r\n").split(/\s+/, 2)
    Console.logger.info(self, "Received #{method} #{path}")

    if file = FILES.get(path)
      Console.logger.info(self, "Serving #{path}")
      connection.write(file.read)
      file.close
    else
      Console.logger.warn(self, "Could not find #{path}")
    end

  rescue => error
    Console.logger.error(self, error)
```

Then we read the file and write it back to the client.

```ruby
    stream = Async::IO::Stream.new(connection)
    method, path = stream.read_until("\r\n").split(/\s+/, 2)
    Console.logger.info(self, "Received #{method} #{path}")

    if file = FILES.get(path)
      Console.logger.info(self, "Serving #{path}")
      connection.write(file.read)
      file.close
    else
      Console.logger.warn(self, "Could not find #{path}")
    end

  rescue => error
    Console.logger.error(self, error)
  ensure
    connection.close
  end
end
```

Finally, we close the connection, to indicate the completed response.



So at a high level, *the client connects and *writes a request line, *the server reads that request line and *fetches the file, then it *writes the file to the client which *reads everything until the *connection is closed.

Can only handle one request per connection.

接続ごとに処理できるリクエストは1つだけです。

A limitation of this design, is that only one request can be processed per connection. Closing the connection is used to indicate the end of the response body.

No way to detect errors.

エラーを検出する方法がありません。

In addition, there is no way to detect errors - for example if the file can't be found, or if the network connection fails, the response body may be missing or truncated, and this can't be detected.

HTTP/1.0

(5:40) The first official release, HTTP/1.0 was a significant improvement and aimed to fix some of these limitations. The main RFC is about 18,000 words long.

Released in 1996 with support for new HTTP methods like POST, PUT, and DELETE.

1996 年にリリースされ、POST、PUT、DELETE などの新しい HTTP メソッドがサポートされました。

It was released in 1996 with support for new HTTP methods like POST, PUT and DELETE, for handling HTML forms and file uploads.

Included a response line with a status code.

ステータス コードを含む応答行が含まれています。

It also included a response line with a status code to indicate the nature of the response…

```
HTTP/1.0 200 OK
```

For example, 200 OK which is used when the request is successful…

```
HTTP/1.0 404 Not Found
```

Or 404 Not Found if the requested path doesn’t exist.

```
HTTP/1.0 418 I'm a teapot
```

Or even 418 I’m a teapot if you accidentally send your request to the wrong kitchen appliance.

Introduced support for headers, allowing for more complex requests and responses.

ヘッダーのサポートが導入され、より複雑なリクエストと応答が可能になりました。

HTTP/1.0 introduced support for request and response headers, allowing for more complex semantics like content-type and content-length.

Added support for multimedia and other types of content through the use of MIME types.

MIME タイプの使用により、マルチメディアおよびその他のタイプのコンテンツのサポートが追加されました。

…and with these new headers, supporting other formats of content became possible using MIME types, so web sites could have embedded images, audio and video.

The first version of HTTP to gain significant adoption and use in the industry.

業界で大幅に採用され、使用されるようになった最初の HTTP バージョン。

It was also the first version of HTTP to gain significant adoption and use in the industry.

HTTP/1.0 Client

So let’s take a look at the changes required to make an HTTP/1.0 client.

```ruby
#!/usr/bin/env ruby

require 'async'
require 'async/io'
require 'async/io/stream'

Sync do
  endpoint = Async::IO::Endpoint.tcp('localhost', 8010)

  ARGV.each do |path|
    endpoint.connect do |connection|
      stream = Async::IO::Stream.new(connection)
      stream.write("GET #{path} HTTP/1.0\r\n")
      stream.write("Accept: text/html\r\n")
      stream.write("\r\n")
      stream.flush

      response_line = stream.read_until("\r\n")
      version, status, reason = response_line.split(' ', 3)
```

Here is the updated client code.

When we make a request, we include the protocol version. Also, different request methods could be used here.

We can also include a list of headers, these are name-value pairs which indicate something about the request, in this case we are telling the server that we'd accept a text/html response format.

```
      stream = Async::IO::Stream.new(connection)
      stream.write("GET #{path} HTTP/1.0\r\n")
      stream.write("Accept: text/html\r\n")
      stream.write("\r\n")
      stream.flush

      response_line = stream.read_until("\r\n")
      version, status, reason = response_line.split(' ', 3)
      Console.logger.info(self, "Received #{version} #{status} #{reason}")

      while line = stream.read_until("\r\n")
        break if line.empty?
        name, value = line.split(/:\s*/, 2)
        Console.logger.info(self, "Header #{name}: #{value}")
      end

      puts stream.read
    end
  end
end
```

After submitting the request, the client must wait for a response line from the server.

```
      stream = Async::IO::Stream.new(connection)
      stream.write("GET #{path} HTTP/1.0\r\n")
      stream.write("Accept: text/html\r\n")
      stream.write("\r\n")
      stream.flush

      response_line = stream.read_until("\r\n")
      version, status, reason = response_line.split(' ', 3)
      Console.logger.info(self, "Received #{version} #{status} #{reason}")

      while line = stream.read_until("\r\n")
        break if line.empty?
        name, value = line.split(/:\s*/, 2)
        Console.logger.info(self, "Header #{name}: #{value}")
      end

      puts stream.read
    end
  end
end
```

Most importantly, this line includes the status of the response.

```
      stream = Async::IO::Stream.new(connection)
      stream.write("GET #{path} HTTP/1.0\r\n")
      stream.write("Accept: text/html\r\n")
      stream.write("\r\n")
      stream.flush

      response_line = stream.read_until("\r\n")
      version, status, reason = response_line.split(' ', 3)
      Console.logger.info(self, "Received #{version} #{status} #{reason}")

      while line = stream.read_until("\r\n")
        break if line.empty?
        name, value = line.split(/:\s*/, 2)
        Console.logger.info(self, "Header #{name}: #{value}")
      end

      puts stream.read
    end
  end
end
```

Next, the client must read any headers generated as part of the response.

```ruby
        stream = Async::IO::Stream.new(connection)
        stream.write("GET #{path} HTTP/1.0\r\n")
        stream.write("Accept: text/html\r\n")
        stream.write("\r\n")
        stream.flush

        response_line = stream.read_until("\r\n")
        version, status, reason = response_line.split(' ', 3)
        Console.logger.info(self, "Received #{version} #{status} #{reason}")

        while line = stream.read_until("\r\n")
          break if line.empty?
          name, value = line.split(/:\s*/, 2)
          Console.logger.info(self, "Header #{name}: #{value}")
        end

        puts stream.read
      end
    end
  end
```

Finally, the client reads the response body until the connection is closed.

HTTP/1.0 Server

Now let's take a look at the changes required to the server.

```ruby
#!/usr/bin/env ruby

require 'async'
require 'async/io'
require 'async/io/stream'
require_relative '../files'

Sync do
  endpoint = Async::IO::Endpoint.tcp('localhost', 8010)

  endpoint.accept do |connection|
    stream = Async::IO::Stream.new(connection)
    method, path, version = stream.read_until("\r\n").split(/\s+/, 3)
    Console.logger.info(self, "Received #{method} #{path} #{version}")

    while line = stream.read_until("\r\n")
      break if line.empty?
      name, value = line.split(/:\s*/, 2)
      Console.logger.info(self, "Header #{name}: #{value}")
```

Here is the updated server code.

```ruby
#!/usr/bin/env ruby

require 'async'
require 'async/io'
require 'async/io/stream'
require_relative '../files'

Sync do
  endpoint = Async::IO::Endpoint.tcp('localhost', 8010)

  endpoint.accept do |connection|
    stream = Async::IO::Stream.new(connection)
    method, path, version = stream.read_until("\r\n").split(/\s+/, 3)
    Console.logger.info(self, "Received #{method} #{path} #{version}")

    while line = stream.read_until("\r\n")
      break if line.empty?
      name, value = line.split(/:\s*/, 2)
      Console.logger.info(self, "Header #{name}: #{value}")
```

We extract the method, path and version of the client request.

```ruby
  endpoint = Async::IO::Endpoint.tcp('localhost', 8010)

  endpoint.accept do |connection|
    stream = Async::IO::Stream.new(connection)
    method, path, version = stream.read_until("\r\n").split(/\s+/, 3)
    Console.logger.info(self, "Received #{method} #{path} #{version}")

    while line = stream.read_until("\r\n")
      break if line.empty?
      name, value = line.split(/:\s*/, 2)
      Console.logger.info(self, "Header #{name}: #{value}")
    end

    if file = FILES.get(path)
      Console.logger.info(self, "Serving #{path}")
      stream.write("HTTP/1.0 200 OK\r\n")
      stream.write("Content-Type: text/html\r\n")
      stream.write("\r\n")
      stream.write(file.read)
```

Then, we read any headers the client sends.

```ruby
    method, path, version = stream.read_until("\r\n").split(/\s+/, 3)
    Console.logger.info(self, "Received #{method} #{path} #{version}")

    while line = stream.read_until("\r\n")
      break if line.empty?
      name, value = line.split(/:\s*/, 2)
      Console.logger.info(self, "Header #{name}: #{value}")
    end

    if file = FILES.get(path)
      Console.logger.info(self, "Serving #{path}")
      stream.write("HTTP/1.0 200 OK\r\n")
      stream.write("Content-Type: text/html\r\n")
      stream.write("\r\n")
      stream.write(file.read)
      file.close
    else
      Console.logger.warn(self, "Could not find #{path}")
      stream.write("HTTP/1.0 404 Not Found\r\n")
      stream.write("\r\n")
```

After reading the client request, we can generate the response. Firstly, we write our response line, with a status code.

```
    method, path, version = stream.read_until("\r\n").split(/\s+/, 3)
    Console.logger.info(self, "Received #{method} #{path} #{version}")

    while line = stream.read_until("\r\n")
      break if line.empty?
      name, value = line.split(/:\s*/, 2)
      Console.logger.info(self, "Header #{name}: #{value}")
    end

    if file = FILES.get(path)
      Console.logger.info(self, "Serving #{path}")
      stream.write("HTTP/1.0 200 OK\r\n")
      stream.write("Content-Type: text/html\r\n")
      stream.write("\r\n")
      stream.write(file.read)
      file.close
    else
      Console.logger.warn(self, "Could not find #{path}")
      stream.write("HTTP/1.0 404 Not Found\r\n")
      stream.write("\r\n")
```

Then we write a response header with a content type, this indicates that the document is HTML.

```
    method, path, version = stream.read_until("\r\n").split(/\s+/, 3)
    Console.logger.info(self, "Received #{method} #{path} #{version}")

    while line = stream.read_until("\r\n")
      break if line.empty?
      name, value = line.split(/:\s*/, 2)
      Console.logger.info(self, "Header #{name}: #{value}")
    end

    if file = FILES.get(path)
      Console.logger.info(self, "Serving #{path}")
      stream.write("HTTP/1.0 200 OK\r\n")
      stream.write("Content-Type: text/html\r\n")
      stream.write("\r\n")
      stream.write(file.read)
      file.close
    else
      Console.logger.warn(self, "Could not find #{path}")
      stream.write("HTTP/1.0 404 Not Found\r\n")
      stream.write("\r\n")
```

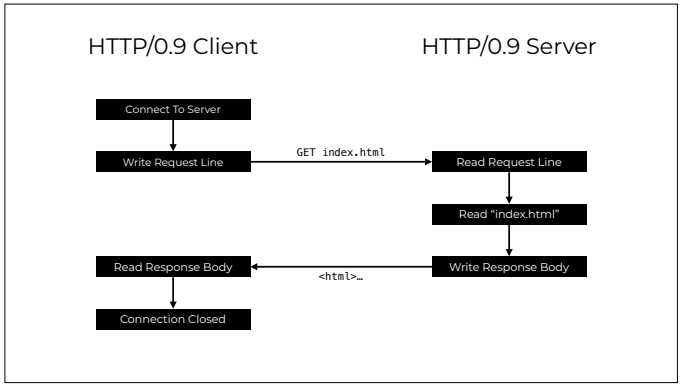
Then we write the response body, the same as before.

```
    end

    if file = FILES.get(path)
      Console.logger.info(self, "Serving #{path}")
      stream.write("HTTP/1.0 200 OK\r\n")
      stream.write("Content-Type: text/html\r\n")
      stream.write("\r\n")
      stream.write(file.read)
      file.close
    else
      Console.logger.warn(self, "Could not find #{path}")
      stream.write("HTTP/1.0 404 Not Found\r\n")
      stream.write("\r\n")
    end

    stream.flush
  rescue => error
    Console.logger.error(self, error)
  ensure
    connection.close
```

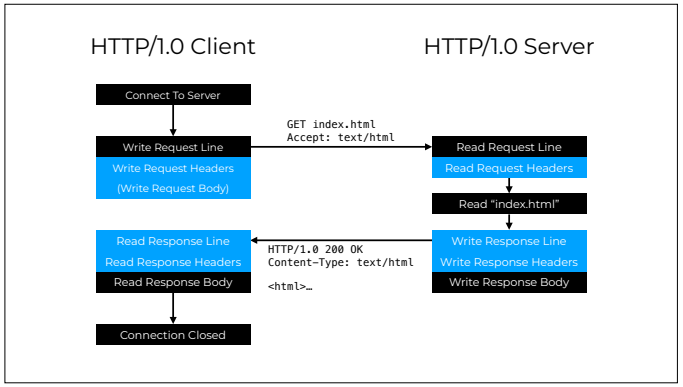
Alternatively if we couldn't find the requested file, we can return a 404 status code.

```
      stream.write("HTTP/1.0 200 OK\r\n")
      stream.write("Content-Type: text/html\r\n")
      stream.write("\r\n")
      stream.write(file.read)
      file.close
    else
      Console.logger.warn(self, "Could not find #{path}")
      stream.write("HTTP/1.0 404 Not Found\r\n")
      stream.write("\r\n")
    end

    stream.flush
  rescue => error
    Console.logger.error(self, error)
  ensure
    connection.close
  end
end
```

Finally, we close the connection.



So, at a high level, this is how our HTTP/0.9 client worked.



And these are the changes introduced by HTTP/1.0 - request headers, response status and response headers.

Still only one request per connection.

まだ、接続ごとにリクエストは 1 つだけです。

However, the official specification still only allowed for one request per connection, which as we mentioned, causes unnecessary network overhead.

HTTP/1.1

(8:30) The next release, HTTP/1.1 tried to address these issues.

Released in 1999 with significant performance improvements.

1999 年にリリースされ、パフォーマンスが大幅に向上しました。

The specification was released in 1999 and updated the network protocol to enable significant performance improvements.

Introduced persistent connections, allowing for multiple requests and responses to be sent over a single connection.

永続的な接続が導入され、単一の接続上で複数のリクエストと応答を送信できるようになりました。

Notably, it introduced support for persistent connections, allowing multiple requests and responses over a single connection.

Added support for chunked transfer encoding, which allowed for more efficient transfer of large files.

チャンク転送エンコーディングのサポートが追加されました。これにより、大きなファイルのより効率的な転送が可能になりました。

It also added support for chunked transfer encoding, which allowed for more efficient transfer of large files.

Became the most widely used version of HTTP and remains in use today.

HTTP の最も広く使用されているバージョンとなり、現在も使用され続けています。

HTTP/1.1 was widely used and remains in use today.

HTTP/1.1 Client

So, let's take a look at the changes required to support persistent connections.

```ruby
#!/usr/bin/env ruby

require 'async'
require 'async/io'
require 'async/io/stream'

Sync do
  endpoint = Async::IO::Endpoint.tcp('localhost', 8011)

  endpoint.connect do |connection|
    stream = Async::IO::Stream.new(connection)
    ARGV.each do |path|
      stream.write("GET #{path} HTTP/1.1\r\n")
      stream.write("Accept: text/html\r\n")
      stream.write("\r\n")
      stream.flush

      response_line = stream.read_until("\r\n")
      version, status, reason = response_line.split(' ', 3)
```

Our client code is largely the same as before, with a few small changes.

```ruby
#!/usr/bin/env ruby

require 'async'
require 'async/io'
require 'async/io/stream'

Sync do
  endpoint = Async::IO::Endpoint.tcp('localhost', 8011)

  endpoint.connect do |connection|
    stream = Async::IO::Stream.new(connection)
    ARGV.each do |path|
      stream.write("GET #{path} HTTP/1.1\r\n")
      stream.write("Accept: text/html\r\n")
      stream.write("\r\n")
      stream.flush

      response_line = stream.read_until("\r\n")
      version, status, reason = response_line.split(' ', 3)
```

Instead of making one connection per path, we make a single connection,

```ruby
#!/usr/bin/env ruby

require 'async'
require 'async/io'
require 'async/io/stream'

Sync do
  endpoint = Async::IO::Endpoint.tcp('localhost', 8011)

  endpoint.connect do |connection|
    stream = Async::IO::Stream.new(connection)
    ARGV.each do |path|
      stream.write("GET #{path} HTTP/1.1\r\n")
      stream.write("Accept: text/html\r\n")
      stream.write("\r\n")
      stream.flush

      response_line = stream.read_until("\r\n")
      version, status, reason = response_line.split(' ', 3)
```

…and then make several requests, one for each path, using that same connection.

```ruby
      Console.logger.info(self, "Received #{version} #{status} #{reason}")

      length = nil

      while line = stream.read_until("\r\n")
        break if line.empty?

        name, value = line.split(/:\s+/, 2)
        if name.downcase == "content-length"
          length = Integer(value)
        end
        Console.logger.info(self, "Header #{name}: #{value}")
      end

      if length
        if length > 0
          puts stream.read(length)
        end
      else
```

When reading a response from the server, the server can indicate the content length of the response body using a response header.

```ruby
        break if line.empty?

        name, value = line.split(/:\s+/, 2)
        if name.downcase == "content-length"
          length = Integer(value)
        end
        Console.logger.info(self, "Header #{name}: #{value}")
      end

      if length
        if length > 0
          puts stream.read(length)
        end
      else
        puts stream.read
      end
    end
  end
end
```

In that case, if the length is known on the client,

```
        break if line.empty?

        name, value = line.split(/:\s+/, 2)
        if name.downcase == "content-length"
          length = Integer(value)
        end
        Console.logger.info(self, "Header #{name}: #{value}")
      end

      if length
        if length > 0
          puts stream.read(length)
        end
      else
        puts stream.read
      end
    end
  end
end
```

The client reads exactly that amount, and can then make another request and can then reuse the same connection to make another request.

HTTP/1.1 Server

(10:00) The server also has very few changes.

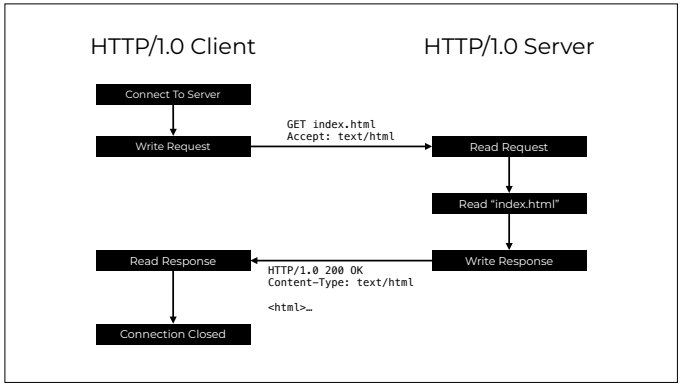
```
#!/usr/bin/env ruby

require 'async'
require 'async/io'
require 'async/io/stream'
require_relative '../files'

Sync do
  endpoint = Async::IO::Endpoint.tcp('localhost', 8011)

  endpoint.accept do |connection|
    stream = Async::IO::Stream.new(connection)
    while request_line = stream.read_until("\r\n")
      method, path, version = request_line.split(/\s+/, 3)
      Console.logger.info(self, "Received #{method} #{path} #{version}")

      while line = stream.read_until("\r\n")
        break if line.empty?
        name, value = line.split(/:\s*/, 2)
```

It’s mostly the same as before.

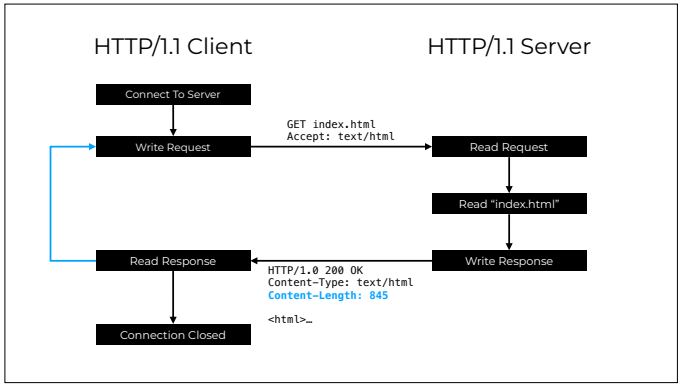
However, when we serve the file, we read the contents of the file into a buffer.

Then, we add a content length header to the response.

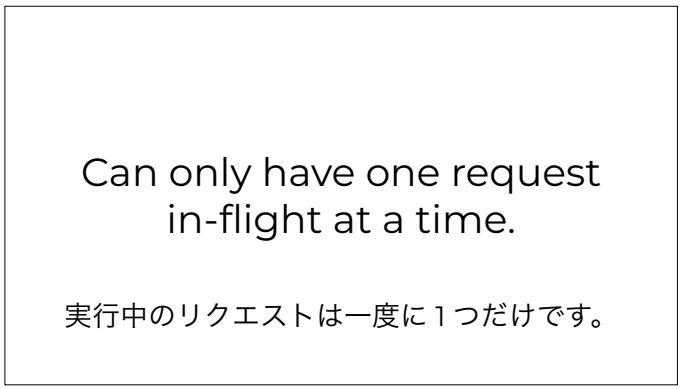
After that, we write the body, and if the connection is not closed, we can read another request.



So, a simplified view of the previous HTTP/1.0 implementation,

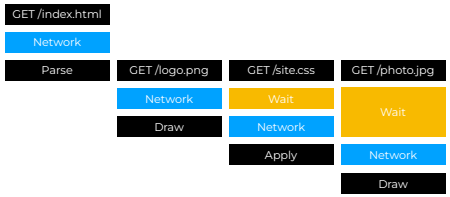


…is extended in HTTP/1.1 such that when the length is known, closing the connection is no longer used to indicate the end of the response body. The connection remains open and can be reused for another request.



However, one big limitation still remains: we can only have one request active at a time - they cannot overlap. So why is this a problem?

HTTP/1 Waiting



On a modern web page, *you have the initial document which often refers to *several other resources. Because you can only send one request at a time, *subsequent requests will have to wait for their turn to use the network connection.

HTTP/2

In order to avoid this kind of waiting, HTTP/2 was introduced and was a significant departure from HTTP/1.

Released in 2015 with a focus on improving performance and reducing latency.

パフォーマンスの向上と遅延の削減に重点を置いて 2015 年にリリースされました。

The HTTP/2 specification was released in 2015 with a focus on improving performance and reducing latency. The main RFC is about 25,000 words long.

Introduced a new binary format, which allowed for more efficient parsing of requests and responses.

新しいバイナリ形式が導入され、リクエストとレスポンスのより効率的な解析が可能になりました。

It introduced a new binary format which allowed for more efficient parsing of requests and responses.

Supports multiplexing of requests and responses, allowing for more efficient use of network resources.

リクエストとレスポンスの多重化をサポートし、ネットワーク リソースをより効率的に使用できるようにします。

It supports concurrent multiplexing of requests and responses on a single TCP connection, allowing for more efficient use of network resources. In other words, you can have several requests active at the same time.

Introduced header compression, which reduced the overhead of sending headers with each request.

ヘッダー圧縮が導入されました。これにより、各リクエストでヘッダーを送信するオーバーヘッドが削減されました。

It also includes HPACK, a specification for header compression which significantly reduces the overhead of transmitting commonly used headers.

Designed to work seamlessly with existing HTTP/1.1 applications and infrastructure.

既存の HTTP/1.1 アプリケーションおよびインフラストラクチャとシームレスに動作するように設計されています。

And while HTTP/2 introduced significant improvements to the protocol, it was designed with the same semantics as HTTP/1, so it could work seamlessly with existing applications and infrastructure.

HTTP/2 Client

So, let's take a look at how to implement an HTTP/2 client.

```ruby
#!/usr/bin/env ruby

require 'async'
require 'async/io/stream'
require 'async/http/endpoint'
require 'protocol/http2/client'

CLIENT_SETTINGS = {
  ::Protocol::HTTP2::Settings::ENABLE_PUSH => 0,
  ::Protocol::HTTP2::Settings::MAXIMUM_FRAME_SIZE => 0x100000,
  ::Protocol::HTTP2::Settings::INITIAL_WINDOW_SIZE => 0x800000,
}

Sync do
  endpoint = Async::HTTP::Endpoint.parse("http://localhost:8020")
  connection = endpoint.connect
  framer = Protocol::HTTP2::Framer.new(connection)
  client = Protocol::HTTP2::Client.new(framer)
```

The implementation of HTTP/2 is actually fairly complex…

```ruby
#!/usr/bin/env ruby

require 'async'
require 'async/io/stream'
require 'async/http/endpoint'
require 'protocol/http2/client'

CLIENT_SETTINGS = {
  ::Protocol::HTTP2::Settings::ENABLE_PUSH => 0,
  ::Protocol::HTTP2::Settings::MAXIMUM_FRAME_SIZE => 0x100000,
  ::Protocol::HTTP2::Settings::INITIAL_WINDOW_SIZE => 0x800000,
}

Sync do
  endpoint = Async::HTTP::Endpoint.parse("http://localhost:8020")
  connection = endpoint.connect
  framer = Protocol::HTTP2::Framer.new(connection)
  client = Protocol::HTTP2::Client.new(framer)
```

…so we are going to use a Ruby gem called Protocol::HTTP2 which implements the binary framing and semantics of the connection handling.

```ruby
#!/usr/bin/env ruby

require 'async'
require 'async/io/stream'
require 'async/http/endpoint'
require 'protocol/http2/client'

CLIENT_SETTINGS = {
  ::Protocol::HTTP2::Settings::ENABLE_PUSH => 0,
  ::Protocol::HTTP2::Settings::MAXIMUM_FRAME_SIZE => 0x100000,
  ::Protocol::HTTP2::Settings::INITIAL_WINDOW_SIZE => 0x800000,
}

Sync do
  endpoint = Async::HTTP::Endpoint.parse("http://localhost:8020")
  connection = endpoint.connect
  framer = Protocol::HTTP2::Framer.new(connection)
  client = Protocol::HTTP2::Client.new(framer)
```

HTTP/2 include specific settings for negotiating details of the connection, such as frame size and flow control.

```ruby
  ::Protocol::HTTP2::Settings::MAXIMUM_FRAME_SIZE => 0x100000,
  ::Protocol::HTTP2::Settings::INITIAL_WINDOW_SIZE => 0x800000,
}

Sync do
  endpoint = Async::HTTP::Endpoint.parse("http://localhost:8020")
  connection = endpoint.connect
  framer = Protocol::HTTP2::Framer.new(connection)
  client = Protocol::HTTP2::Client.new(framer)

  client.send_connection_preface(CLIENT_SETTINGS)

  ARGV.each do |path|
    stream = client.create_stream

    headers = [
      [":scheme", endpoint.scheme],
      [":method", "GET"],
      [":authority", "localhost"],
      [":path", path]
```

As before, the client makes a connection using TCP.

```
  ::Protocol::HTTP2::Settings::MAXIMUM_FRAME_SIZE => 0x100000,
  ::Protocol::HTTP2::Settings::INITIAL_WINDOW_SIZE => 0x800000,
}

Sync do
  endpoint = Async::HTTP::Endpoint.parse("http://localhost:8020")
  connection = endpoint.connect
  framer = Protocol::HTTP2::Framer.new(connection)
  client = Protocol::HTTP2::Client.new(framer)

  client.send_connection_preface(CLIENT_SETTINGS)

  ARGV.each do |path|
    stream = client.create_stream

    headers = [
      [":scheme", endpoint.scheme],
      [":method", "GET"],
      [":authority", "localhost"],
      [":path", path]
```

We wrap this connection in a framer, which can read and write binary frames.

```
  ::Protocol::HTTP2::Settings::MAXIMUM_FRAME_SIZE => 0x100000,
  ::Protocol::HTTP2::Settings::INITIAL_WINDOW_SIZE => 0x800000,
}

Sync do
  endpoint = Async::HTTP::Endpoint.parse("http://localhost:8020")
  connection = endpoint.connect
  framer = Protocol::HTTP2::Framer.new(connection)
  client = Protocol::HTTP2::Client.new(framer)

  client.send_connection_preface(CLIENT_SETTINGS)

  ARGV.each do |path|
    stream = client.create_stream

    headers = [
      [":scheme", endpoint.scheme],
      [":method", "GET"],
      [":authority", "localhost"],
      [":path", path]
```

We then create a client which will process those frames and manage the connection state.

```
  ::Protocol::HTTP2::Settings::MAXIMUM_FRAME_SIZE => 0x100000,
  ::Protocol::HTTP2::Settings::INITIAL_WINDOW_SIZE => 0x800000,
}

Sync do
  endpoint = Async::HTTP::Endpoint.parse("http://localhost:8020")
  connection = endpoint.connect
  framer = Protocol::HTTP2::Framer.new(connection)
  client = Protocol::HTTP2::Client.new(framer)

  client.send_connection_preface(CLIENT_SETTINGS)

  ARGV.each do |path|
    stream = client.create_stream

    headers = [
      [":scheme", endpoint.scheme],
      [":method", "GET"],
      [":authority", "localhost"],
      [":path", path]
```

To start the connection, the client must send a connection preface which includes the settings.

```ruby
ARGV.each do |path|
  stream = client.create_stream

  headers = [
    [":scheme", endpoint.scheme],
    [":method", "GET"],
    [":authority", "localhost"],
    [":path", path],
    ["accept", "*/*"],
  ]

  stream.send_headers(nil, headers, Protocol::HTTP2::END_STREAM)

  def stream.process_headers(frame)
    headers = super
    Console.logger.info(self, "Received #{headers}")
  end

  def stream.process_data(frame)
```

Then, for each path we want to request, we create a stream

```ruby
ARGV.each do |path|
  stream = client.create_stream

  headers = [
    [":scheme", endpoint.scheme],
    [":method", "GET"],
    [":authority", "localhost"],
    [":path", path],
    ["accept", "*/*"],
  ]

  stream.send_headers(nil, headers, Protocol::HTTP2::END_STREAM)

  def stream.process_headers(frame)
    headers = super
    Console.logger.info(self, "Received #{headers}")
  end

  def stream.process_data(frame)
```

“Request Line” Pseudo Headers

HTTP/2 no longer has a request line, all those fields, including scheme, method, authority and path are now included in the headers. These special request headers start with a colon and are referred to as pseudo-headers.

```ruby
ARGV.each do |path|
  stream = client.create_stream

  headers = [
    [":scheme", endpoint.scheme],
    [":method", "GET"],
    [":authority", "localhost"],
    [":path", path],
    ["accept", "*/*"],
  ]

  stream.send_headers(nil, headers, Protocol::HTTP2::END_STREAM)

  def stream.process_headers(frame)
    headers = super
    Console.logger.info(self, "Received #{headers}")
  end

  def stream.process_data(frame)
```

We encode those headers into a binary frame using HPACK, and send it on the stream, which starts the request.

```
    stream.send_headers(nil, headers, Protocol::HTTP2::END_STREAM)

    def stream.process_headers(frame)
      headers = super
      Console.logger.info(self, "Received #{headers}")
    end

    def stream.process_data(frame)
      if data = super
        $stdout.write(data)
      end
    end

    until stream.closed?
      client.read_frame
    end
  end

  client.send_goaway
```

HEADERS frame callback

Then, we add a callback for the response headers,

```
    stream.send_headers(nil, headers, Protocol::HTTP2::END_STREAM)

    def stream.process_headers(frame)
      headers = super
      Console.logger.info(self, "Received #{headers}")
    end

    def stream.process_data(frame)
      if data = super
        $stdout.write(data)
      end
    end

    until stream.closed?
      client.read_frame
    end
  end

  client.send_goaway
```

DATA frame callback

And the response body data.

```
    stream.send_headers(nil, headers, Protocol::HTTP2::END_STREAM)

    def stream.process_headers(frame)
      headers = super
      Console.logger.info(self, "Received #{headers}")
    end

    def stream.process_data(frame)
      if data = super
        $stdout.write(data)
      end
    end

    until stream.closed?
      client.read_frame
    end
  end

  client.send_goaway
```

Finally, we read frames until the stream is closed. This means that the response has been received completely.

HTTP/2 Server

Now let's take a look at the server implementation.

```ruby
#!/usr/bin/env ruby

require_relative '../files'

require 'async'
require 'async/io/stream'
require 'async/http/endpoint'
require 'protocol/http2/client'

SERVER_SETTINGS = {
  ::Protocol::HTTP2::Settings::MAXIMUM_CONCURRENT_STREAMS => 128,
  ::Protocol::HTTP2::Settings::MAXIMUM_FRAME_SIZE => 0x100000,
  ::Protocol::HTTP2::Settings::INITIAL_WINDOW_SIZE => 0x800000,
  ::Protocol::HTTP2::Settings::ENABLE_CONNECT_PROTOCOL => 1,
}

Sync do
  endpoint = Async::HTTP::Endpoint.parse("http://localhost:8020")
  endpoint.accept do |connection|
```

It's totally different from the HTTP/1 implementation, but the actual request/response semantics are similar.

```ruby
#!/usr/bin/env ruby

require_relative '../files'

require 'async'
require 'async/io/stream'
require 'async/http/endpoint'
require 'protocol/http2/client'

SERVER_SETTINGS = {
  ::Protocol::HTTP2::Settings::MAXIMUM_CONCURRENT_STREAMS => 128,
  ::Protocol::HTTP2::Settings::MAXIMUM_FRAME_SIZE => 0x100000,
  ::Protocol::HTTP2::Settings::INITIAL_WINDOW_SIZE => 0x800000,
  ::Protocol::HTTP2::Settings::ENABLE_CONNECT_PROTOCOL => 1,
}

Sync do
  endpoint = Async::HTTP::Endpoint.parse("http://localhost:8020")
  endpoint.accept do |connection|
```

Like the client, the server also specifies settings, like the maximum number of concurrent streams.

```ruby
Sync do
  endpoint = Async::HTTP::Endpoint.parse("http://localhost:8020")
  endpoint.accept do |connection|
    framer = Protocol::HTTP2::Framer.new(connection)
    server = Protocol::HTTP2::Server.new(framer)

    server.read_connection_preface(SERVER_SETTINGS)

    def server.accept_stream(stream_id)
      super.tap do |stream|
        def stream.process_headers(frame)
          headers = super.to_h
          Console.logger.info(self, "Received #{headers}")

          path = headers[':path']

          if file = FILES.get(path)
            Console.logger.info(self, "Serving #{path}")
            self.send_headers(nil, [
```

As before, we accept incoming connections.

```ruby
Sync do
  endpoint = Async::HTTP::Endpoint.parse("http://localhost:8020")
  endpoint.accept do |connection|
    framer = Protocol::HTTP2::Framer.new(connection)
    server = Protocol::HTTP2::Server.new(framer)

    server.read_connection_preface(SERVER_SETTINGS)

    def server.accept_stream(stream_id)
      super.tap do |stream|
        def stream.process_headers(frame)
          headers = super.to_h
          Console.logger.info(self, "Received #{headers}")

          path = headers[':path']

          if file = FILES.get(path)
            Console.logger.info(self, "Serving #{path}")
            self.send_headers(nil, [
```

We create a framer wrapping the connection.

```ruby
Sync do
  endpoint = Async::HTTP::Endpoint.parse("http://localhost:8020")
  endpoint.accept do |connection|
    framer = Protocol::HTTP2::Framer.new(connection)
    server = Protocol::HTTP2::Server.new(framer)

    server.read_connection_preface(SERVER_SETTINGS)

    def server.accept_stream(stream_id)
      super.tap do |stream|
        def stream.process_headers(frame)
          headers = super.to_h
          Console.logger.info(self, "Received #{headers}")

          path = headers[':path']

          if file = FILES.get(path)
            Console.logger.info(self, "Serving #{path}")
            self.send_headers(nil, [
```

Then we create a server instance which manages the state of the connection.

```ruby
Sync do
  endpoint = Async::HTTP::Endpoint.parse("http://localhost:8020")
  endpoint.accept do |connection|
    framer = Protocol::HTTP2::Framer.new(connection)
    server = Protocol::HTTP2::Server.new(framer)

    server.read_connection_preface(SERVER_SETTINGS)

    def server.accept_stream(stream_id)
      super.tap do |stream|
        def stream.process_headers(frame)
          headers = super.to_h
          Console.logger.info(self, "Received #{headers}")

          path = headers[':path']

          if file = FILES.get(path)
            Console.logger.info(self, "Serving #{path}")
            self.send_headers(nil, [
```

The server must read the incoming connection preface and also sends it's settings to the client.

```ruby
    def server.accept_stream(stream_id)
      super.tap do |stream|
        def stream.process_headers(frame)
          headers = super.to_h
          Console.logger.info(self, "Received #{headers}")

          path = headers[':path']

          if file = FILES.get(path)
            Console.logger.info(self, "Serving #{path}")
            self.send_headers(nil, [
              [":status", "200"],
              ["content-type", "text/html"],
            ])

            self.send_data(file.read, Protocol::HTTP2::END_STREAM)
            file.close
          else
            Console.logger.warn(self, "Could not find #{path}")
```

Then, we add a accept_stream callback.

```ruby
    def server.accept_stream(stream_id)
      super.tap do |stream|
        def stream.process_headers(frame)
          headers = super.to_h
          Console.logger.info(self, "Received #{headers}")

          path = headers[':path']

          if file = FILES.get(path)
            Console.logger.info(self, "Serving #{path}")
            self.send_headers(nil, [
              [":status", "200"],
              ["content-type", "text/html"],
            ])

            self.send_data(file.read, Protocol::HTTP2::END_STREAM)
            file.close
          else
            Console.logger.warn(self, "Could not find #{path}")
```

To that stream, we add a process headers callback, which is invoked when the client stream sends the request headers.

```
def server.accept_stream(stream_id)
  super.tap do |stream|
    def stream.process_headers(frame)
      headers = super.to_h
      Console.logger.info(self, "Received #{headers}")

      path = headers[':path']

      if file = FILES.get(path)
        Console.logger.info(self, "Serving #{path}")
        self.send_headers(nil, [
          [":status", "200"],
          ["content-type", "text/html"],
        ])

        self.send_data(file.read, Protocol::HTTP2::END_STREAM)
        file.close
      else
        Console.logger.warn(self, "Could not find #{path}")
```

We extract the path pseudo header.

```
def server.accept_stream(stream_id)
  super.tap do |stream|
    def stream.process_headers(frame)
      headers = super.to_h
      Console.logger.info(self, "Received #{headers}")

      path = headers[':path']

      if file = FILES.get(path)
        Console.logger.info(self, "Serving #{path}")
        self.send_headers(nil, [
          [":status", "200"],
          ["content-type", "text/html"],
        ])

        self.send_data(file.read, Protocol::HTTP2::END_STREAM)
        file.close
      else
        Console.logger.warn(self, "Could not find #{path}")
```

We read the file as before.

```
def server.accept_stream(stream_id)
  super.tap do |stream|
    def stream.process_headers(frame)
      headers = super.to_h
      Console.logger.info(self, "Received #{headers}")

      path = headers[':path']

      if file = FILES.get(path)
        Console.logger.info(self, "Serving #{path}")
        self.send_headers(nil, [
          [":status", "200"],
          ["content-type", "text/html"],
        ])

        self.send_data(file.read, Protocol::HTTP2::END_STREAM)
        file.close
      else
        Console.logger.warn(self, "Could not find #{path}")
```

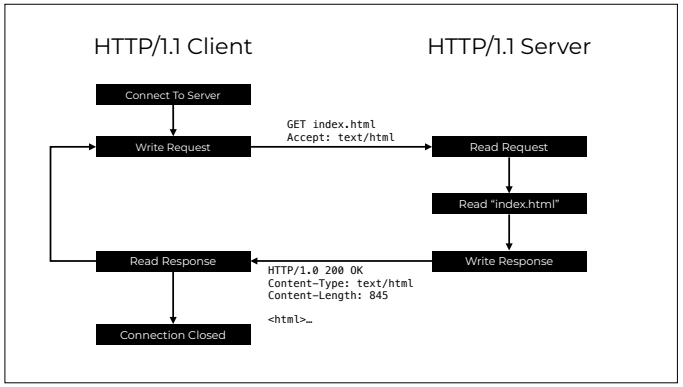
We begin sending the response…

```ruby
def server.accept_stream(stream_id)
  super.tap do |stream|
    def stream.process_headers(frame)
      headers = super.to_h
      Console.logger.info(self, "Received #{headers}")

      path = headers[':path']

      if file = FILES.get(path)
        Console.logger.info(self, "Serving #{path}")
        self.send_headers(nil, [
          [":status", "200"],
          ["content-type", "text/html"],
        ])

        self.send_data(file.read, Protocol::HTTP2::END_STREAM)
        file.close
      else
        Console.logger.warn(self, "Could not find #{path}")
```

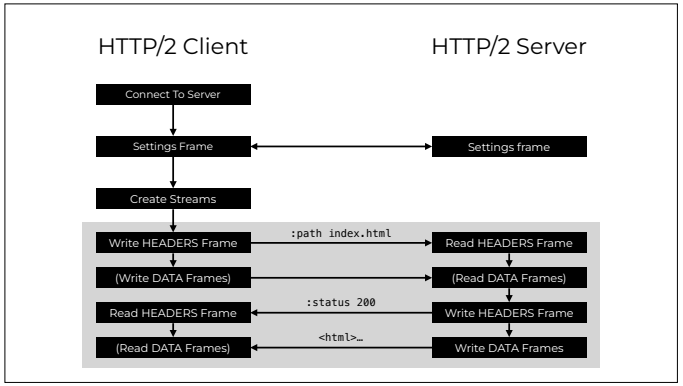
Note that we include a status code pseudo header - there is no response line like there is in HTTP/1.

```ruby
def server.accept_stream(stream_id)
  super.tap do |stream|
    def stream.process_headers(frame)
      headers = super.to_h
      Console.logger.info(self, "Received #{headers}")

      path = headers[':path']

      if file = FILES.get(path)
        Console.logger.info(self, "Serving #{path}")
        self.send_headers(nil, [
          [":status", "200"],
          ["content-type", "text/html"],
        ])

        self.send_data(file.read, Protocol::HTTP2::END_STREAM)
        file.close
      else
        Console.logger.warn(self, "Could not find #{path}")
```

Then we send the response body data.



So, let's compare the high level behaviour - this was the HTTP/1.1 client and server.



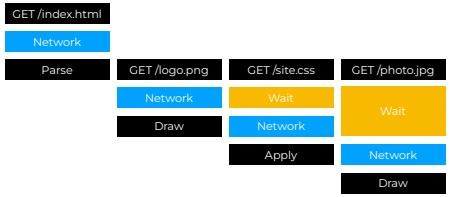
HTTP/2 moves all of the request/response handling into concurrent multiplexed streams. Each stream represents a single request and response.

We can have multiple request in-flight at a time.

実行中に一度に複数のリクエストを行うことができます。

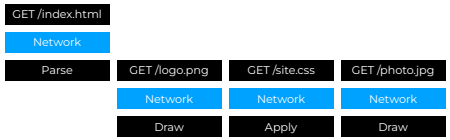
Because those streams operate independently over a shared connection, we can now have multiple requests in-flight at the same time.

HTTP/1 Waiting



So previously we had to send requests one at a time with HTTP/1.

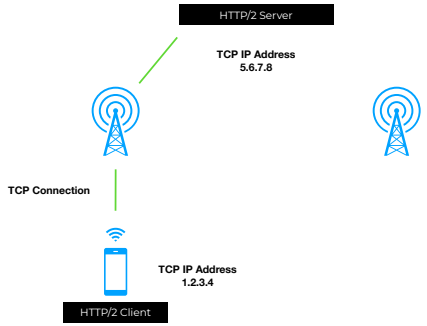
HTTP/2 Multiplexing



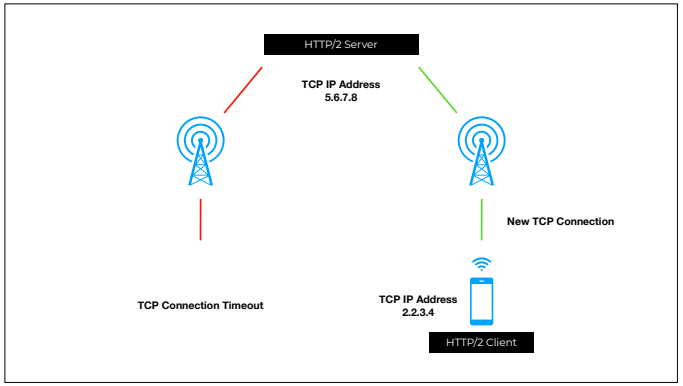
But in HTTP/2, we can start all those requests independently on the same connection.

Multiplexed streams over TCP performs poorly over cellular networks.

However, because they all share the same TCP connection, if the connection experiences disruption or packet loss, all streams will be blocked until the TCP connection recovers or reconnects.



As an example, consider a cellphone moving between two different networks.



The *original TCP connection will be lost, and a *new TCP connection must be established. For things like streaming audio and video, this can be a problem, as the data stream will be interrupted.

HTTP/3

(20:00) HTTP/3 is the latest specification, and was designed to solve this problem.

Released in 2020 with a focus on improving performance and security over unreliable networks.

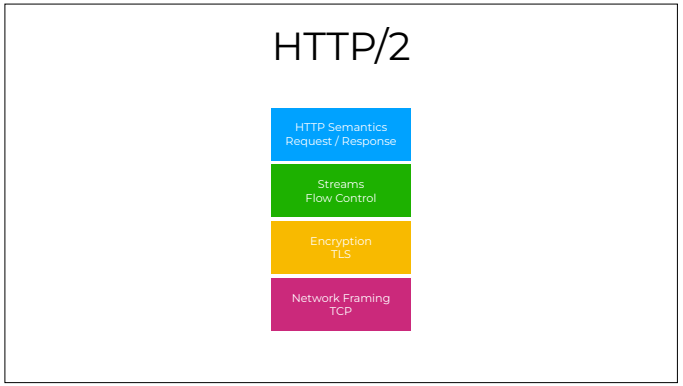
信頼性の低いネットワーク上でのパフォーマンスとセキュリティの向上に焦点を当てて、2020 年にリリースされました。

It was released in 2020 with a focus on improving performance and security over unreliable networks.

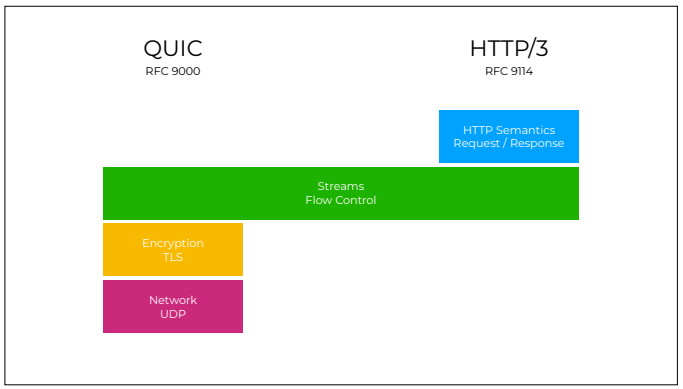
Based on the QUIC transport protocol, which provides a number of performance and security improvements over TCP.

QUIC トランスポート プロトコルに基づいており、TCP に比べてパフォーマンスとセキュリティが数多く向上しています。

HTTP/2 was based on TCP, which as we discussed, can perform poorly when encountering packet loss or connection failure. In order to address this, the QUIC transport protocol was developed, which uses UDP instead.



HTTP/2 can be broken down into 4 core areas: *HTTP Semantics, *Streams, *Encryption and *Network Framing.



HTTP/3 is actually two specifications: QUIC which provides robust, encrypted, multiplexed streams, and HTTP/3, mapping the abstract HTTP semantics to QUIC streams. The main RFCs for QUIC and HTTP/3 are 57,000 and 21,000 words long, respectively.

Designed to reduce the latency of connections and improve the reliability of data transfer.

接続の待ち時間を短縮し、データ転送の信頼性を向上させるように設計されています。

By developing a new network protocol, priority could be given to implementation choices that reduce latency and improve the reliability of data transfer over unreliable networks.

Still in the early stages of adoption, but expected to gain wider use over time.

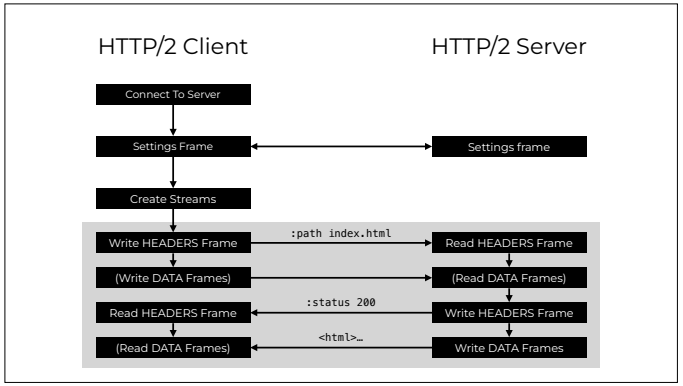
まだ導入の初期段階にありますが、時間の経過とともに広く使用されるようになることが予想されます。

QUIC is still in the early stages of adoption, but it's a significant improvement over HTTP/2, both in terms of design and performance, and it is expected to gain wider use over time.

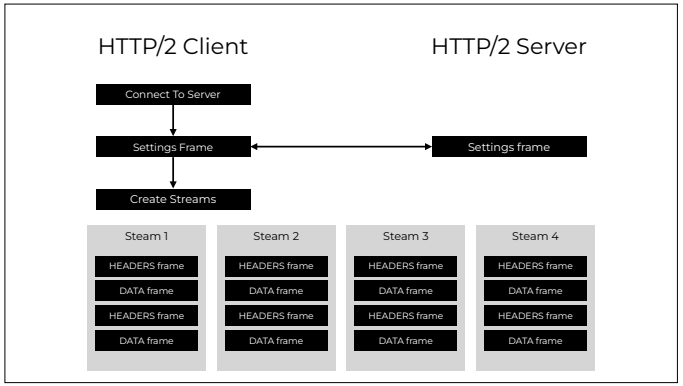
HTTP/3 Client & Server

申し訳ありませんが、HTTP/3 クライアントとサーバーはまだ実装されていません。

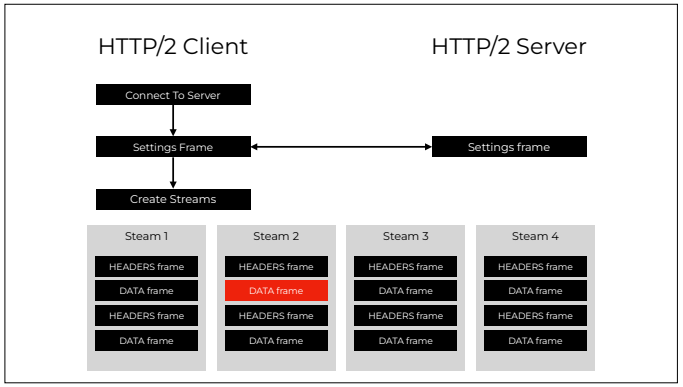
So, I'm a little bit disappointed because I can't present a Ruby HTTP/3 client or server today. I've been working on it for several months but despite my best efforts, it's not ready yet. I will share my progress updates online, so please follow me if you are interested in that.



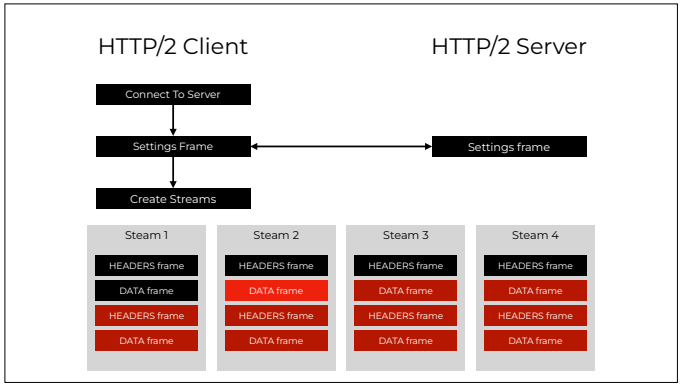
However, we can compare the high level behaviour of HTTP/2 and HTTP/3 to understand the improvements and why they are important. Let's investigate what happens when we have several concurrent streams.



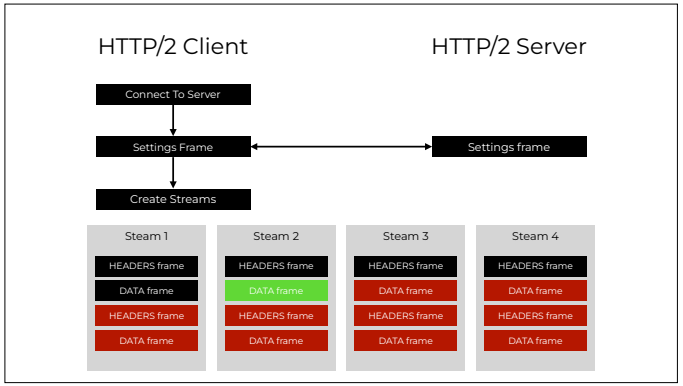
In this example, we have 4 streams, each sending a request and reading a response over HTTP/2.



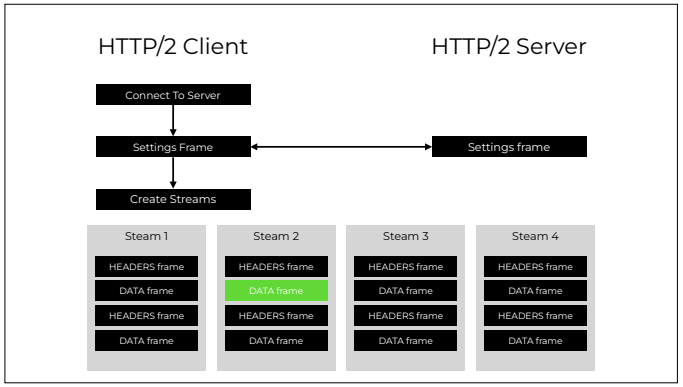
However if the TCP connection experiences packet loss, at least one frame will be interrupted.



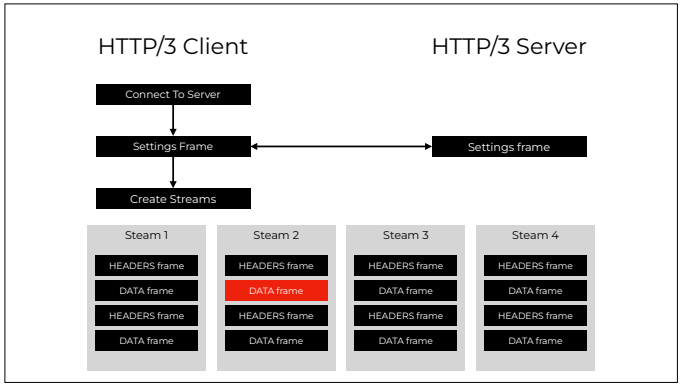
Unfortunately since all streams are sharing the same TCP connection, all subsequent frames, across all streams are now blocked, until TCP recovers the missing data. This is called “Head of line blocking”.



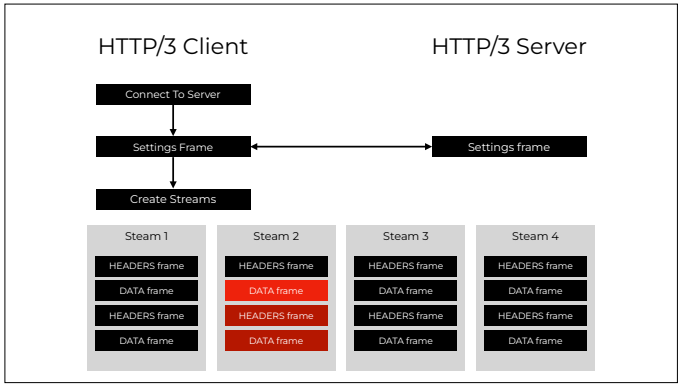
Once TCP recovers, all the streams can continue.



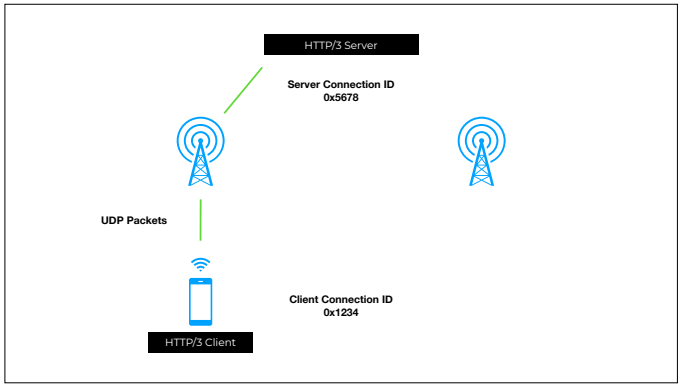
Even though the streams themselves are mostly independent of each other, because they share a single connection, if that connection is interrupted, all streams will be interrupted too.



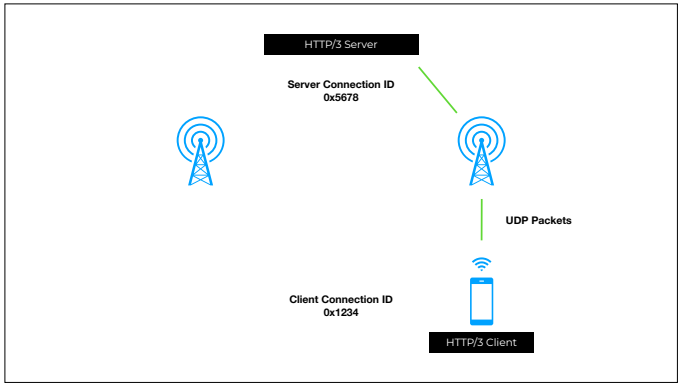
In contrast, when a QUIC stream experiences packet loss, because the streams are sending and receiving packets independently using UDP,



Only the subsequent frames of that specific stream are blocked. The other streams can continue to send and receive UDP packets.



And in comparison to before, while a TCP connection is not capable of migrating between networks, a QUIC connection uses UDP packets with a more robust mechanism for identifying the client and server,

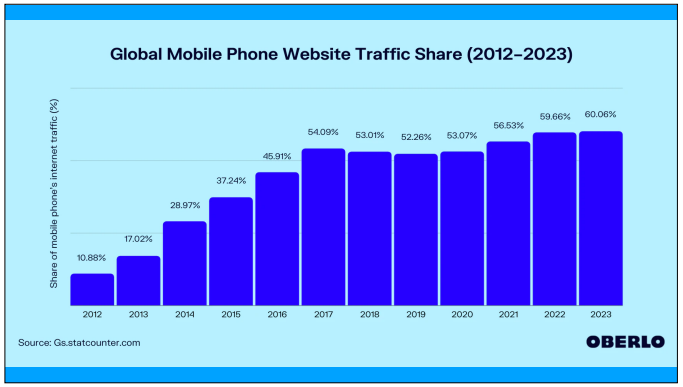


So when you move your mobile device to a different network, the connection will continue without interruption.

Why is connection recovery important?

接続回復が重要なのはなぜですか?

So maybe you are wondering, why is connection recovery important?



As of 2023, according to this data, 60% of website traffic is generated by mobile phones. Mobile phones often move between networks. So this is a very useful feature of QUIC.

Where does Asynchronous Ruby fit into all this?

非同期 Ruby はこれらすべてのどこに当てはまるのでしょうか?

So, we have talked about the evolution of HTTP, but where does Asynchronous Ruby fit into all this?

Are existing HTTP adapters good enough?

既存の HTTP アダプターは十分ですか?

Maybe we can ask a different question: are existing Ruby HTTP adapters good enough?

Very few adapters support HTTP/2+

HTTP/2+ をサポートするアダプターはほとんどありません

Well, it turns out, very few Ruby HTTP clients or servers support HTTP/2 or later. This limits the ability of Ruby applications to take advantage of these evolutionary improvements to HTTP.

Multiplexing requests requires a concurrency model.

リクエストを多重化するには同時実行モデルが必要です。

Multiplexing requests requires a concurrency model. To solve this problem, I introduced the fiber scheduler which provides a concurrency model well suited handling multiple concurrent streams.

Bi-directional streaming requires explicit design.

双方向ストリーミングには明示的な設計が必要です。

In addition, full bi-directional streaming is hard to implement without an explicit concurrency model. While this might not matter for more “traditional” style request/response systems, an increasing number of interesting real-time web services are becoming available, such as any service that operates on real-time data streams.

Async::HTTP

So, to address these issues, I created Async::HTTP.

Supports HTTP/1 and HTTP/2.

HTTP/1 および HTTP/2 をサポートします。

It supports HTTP/1 and HTTP/2 today,

Planned support for HTTP/3 by the end of the year.

HTTP/3 のサポートは年末までに予定されています。

And we have planned support for HTTP/3 by the end of the year.

Hides all the complexity of the protocol.

プロトコルの複雑さをすべて隠します。

It provides client and server interfaces, and hides all of the complexity of the protocol.

It is a core part of the Falcon web server, so you can host Rack compatible applications in Falcon, but it's using Async::HTTP internally.

Full support for bi-directional streaming.

双方向ストリーミングを完全にサポートします。

It has full support for bi-directional streaming, including WebSockets.

Fully asynchronous request handling.

完全に非同期のリクエスト処理。

It is built on top of Async and provides fully asynchronous request and response handling.

Persistent connections where possible.

可能な場合は永続的な接続。

And it uses an internal protocol-aware connection pool which provides persistent connections where possible.

Async HTTP Client

So, let's take a look at how to write an Async::HTTP client.

```ruby
#!/usr/bin/env ruby

require 'async'
require 'async/http/client'
require 'async/http/endpoint'

Async do |task|
  endpoint = Async::HTTP::Endpoint.parse('http://127.0.0.1:9294')
  client = Async::HTTP::Client.new(endpoint)

  ARGV.each do |path|
    response = client.get(path)
    puts response.read
  ensure
    response.close
  end
end
```

This is the entire source code for the client which supports HTTP/1, HTTP/2, and in the future, HTTP/3.

```ruby
#!/usr/bin/env ruby

require 'async'
require 'async/http/client'
require 'async/http/endpoint'

Async do |task|
  endpoint = Async::HTTP::Endpoint.parse('http://127.0.0.1:9294')
  client = Async::HTTP::Client.new(endpoint)

  ARGV.each do |path|
    response = client.get(path)
    puts response.read
  ensure
    response.close
  end
end
```

We need to include the async/http library.

```ruby
#!/usr/bin/env ruby

require 'async'
require 'async/http/client'
require 'async/http/endpoint'

Async do |task|
  endpoint = Async::HTTP::Endpoint.parse('http://127.0.0.1:9294')
  client = Async::HTTP::Client.new(endpoint)

  ARGV.each do |path|
    response = client.get(path)
    puts response.read
  ensure
    response.close
  end
end
```

We specify the endpoint we want to connect to.

```ruby
#!/usr/bin/env ruby

require 'async'
require 'async/http/client'
require 'async/http/endpoint'

Async do |task|
  endpoint = Async::HTTP::Endpoint.parse('http://127.0.0.1:9294')
  client = Async::HTTP::Client.new(endpoint)

  ARGV.each do |path|
    response = client.get(path)
    puts response.read
  ensure
    response.close
  end
end
```

Then we create a client, which handles all the internal protocol details and functions as a connection pool for persistent connections.

```ruby
#!/usr/bin/env ruby

require 'async'
require 'async/http/client'
require 'async/http/endpoint'

Async do |task|
  endpoint = Async::HTTP::Endpoint.parse('http://127.0.0.1:9294')
  client = Async::HTTP::Client.new(endpoint)

  ARGV.each do |path|
    response = client.get(path)
    puts response.read
  ensure
    response.close
  end
end
```

Then we make the GET request.

```ruby
#!/usr/bin/env ruby

require 'async'
require 'async/http/client'
require 'async/http/endpoint'

Async do |task|
  endpoint = Async::HTTP::Endpoint.parse('http://127.0.0.1:9294')
  client = Async::HTTP::Client.new(endpoint)

  ARGV.each do |path|
    response = client.get(path)
    puts response.read
  ensure
    response.close
  end
end
```

Then we can read the response body. It's very easy to use.

Async HTTP Server

The server is equally simple.

```ruby
#!/usr/bin/env ruby

require 'async'
require 'async/http/server'
require 'async/http/endpoint'
require 'async/http/protocol/response'
require_relative '../files'

Sync do |task|
  app = lambda do |request|
    if file = FILES.get(request.path)
      Protocol::HTTP::Response[200, {}, [file.read]]
    else
      Protocol::HTTP::Response[404, {}, []]
    end
  end

  endpoint = Async::HTTP::Endpoint.parse('http://127.0.0.1:9294')
  server = Async::HTTP::Server.new(app, endpoint)
```

Here is the code which runs the server.

```ruby
#!/usr/bin/env ruby

require 'async'
require 'async/http/server'
require 'async/http/endpoint'
require 'async/http/protocol/response'
require_relative '../files'

Sync do |task|
  app = lambda do |request|
    if file = FILES.get(request.path)
      Protocol::HTTP::Response[200, {}, [file.read]]
    else
      Protocol::HTTP::Response[404, {}, []]
    end
  end

  endpoint = Async::HTTP::Endpoint.parse('http://127.0.0.1:9294')
  server = Async::HTTP::Server.new(app, endpoint)
```

Our application is the same as before, it serves files.

```ruby
Sync do |task|
  app = lambda do |request|
    if file = FILES.get(request.path)
      Protocol::HTTP::Response[200, {}, [file.read]]
    else
      Protocol::HTTP::Response[404, {}, []]
    end
  end

  endpoint = Async::HTTP::Endpoint.parse('http://127.0.0.1:9294')
  server = Async::HTTP::Server.new(app, endpoint)

  server.run
end
```

We specify the endpoint we want to run the server on.

```ruby
Sync do |task|
  app = lambda do |request|
    if file = FILES.get(request.path)
      Protocol::HTTP::Response[200, {}, [file.read]]
    else
      Protocol::HTTP::Response[404, {}, []]
    end
  end

  endpoint = Async::HTTP::Endpoint.parse('http://127.0.0.1:9294')
  server = Async::HTTP::Server.new(app, endpoint)

  server.run
end
```

We create the server,

```ruby
Sync do |task|
  app = lambda do |request|
    if file = FILES.get(request.path)
      Protocol::HTTP::Response[200, {}, [file.read]]
    else
      Protocol::HTTP::Response[404, {}, []]
    end
  end

  endpoint = Async::HTTP::Endpoint.parse('http://127.0.0.1:9294')
  server = Async::HTTP::Server.new(app, endpoint)

  server.run
end
```

And run it.

---

33 years of complexity
hidden in ~10 lines of code.

約 10 行のコードに 33 年分の複雑さが隠されています。

Async::HTTP hides 33 years of complexity in ~10 lines of code. Yet, it provides all of the advanced features that enable new and exciting applications.

Best Practices

ベストプラクティス

So, let's look at three best practices for working with Async::HTTP.

---

1. Use a shared instance.

1. 共有インスタンスを使用します。

My first advice is to use a shared instance that enables persistent connections across your whole application. Since most clients are still using TCP and TLS, making a connection can be slow, so reusing existing connections will improve your performance.

```ruby
#!/usr/bin/env ruby

require 'async'
require 'async/http/internet/instance'

Async do |task|
  internet = Async::HTTP::Internet.instance

  ARGV.each do |url|
    response = internet.get(url)
    puts response.read
  ensure
    response.close
  end
end
```

The simplest way to do this, is to use the Internet instance.

```ruby
#!/usr/bin/env ruby

require 'async'
require 'async/http/internet/instance'

Async do |task|
  internet = Async::HTTP::Internet.instance

  ARGV.each do |url|
    response = internet.get(url)
    puts response.read
  ensure
    response.close
  end
end
```

This instance must be used in an asynchronous context, when it goes out of scope, any open connections will be automatically closed.

```ruby
#!/usr/bin/env ruby

require 'async'
require 'async/http/internet/instance'

Async do |task|
  internet = Async::HTTP::Internet.instance

  ARGV.each do |url|
    response = internet.get(url)
    puts response.read
  ensure
    response.close
  end
end
```

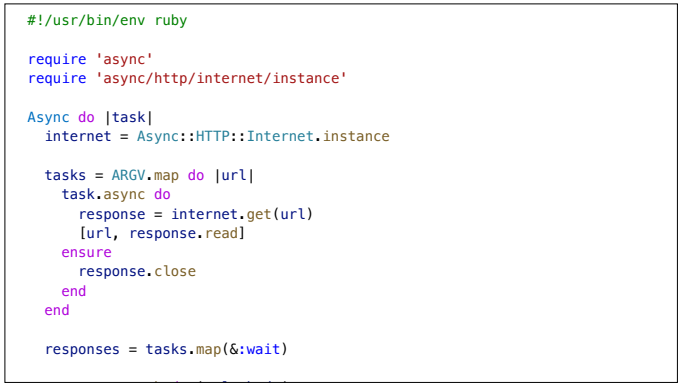
When you make the request, give the full URL, it will automatically figure out the best way to connect and perform the request, and if there is an existing connection, it will be reused.

---

2. Use fan out concurrency.

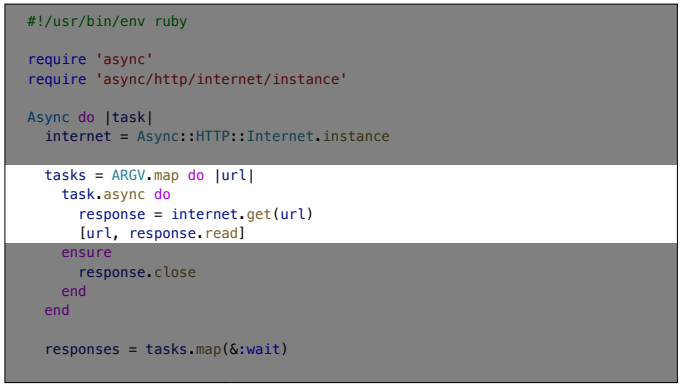
2. ファンアウト同時実行を使用します。

My second advice is to use fan-out concurrency where possible. If you are performing several web requests, running them concurrently will reduce the total latency, either by using several connections, or multiplexing several requests over a single connection.



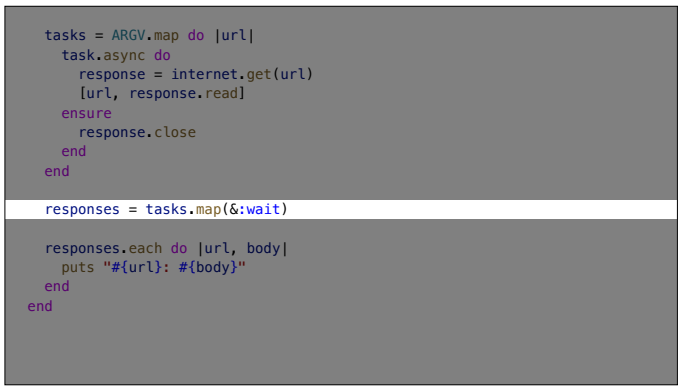
```ruby
#!/usr/bin/env ruby

require 'async'
require 'async/http/internet/instance'

Async do |task|
  internet = Async::HTTP::Internet.instance

  tasks = ARGV.map do |url|
    task.async do
      response = internet.get(url)
      [url, response.read]
    ensure
      response.close
    end
  end

  responses = tasks.map(&:wait)
```

This code is similar to before, but it creates a separate asynchronous task for each request.



```ruby
#!/usr/bin/env ruby

require 'async'
require 'async/http/internet/instance'

Async do |task|
  internet = Async::HTTP::Internet.instance

  tasks = ARGV.map do |url|
    task.async do
      response = internet.get(url)
      [url, response.read]
    ensure
      response.close
    end
  end

  responses = tasks.map(&:wait)
```

For each URL, we make the request and read the response in a child asynchronous task. This allows each request to run concurrently.



```ruby
  tasks = ARGV.map do |url|
    task.async do
      response = internet.get(url)
      [url, response.read]
    ensure
      response.close
    end
  end

  responses = tasks.map(&:wait)

  responses.each do |url, body|
    puts "#{url}: #{body}"
  end
end
```

Then, after we created all the tasks, we wait for them to complete, so we have all the responses available for processing.

3. Embed Async into existing systems.

3. Async を既存のシステムに埋め込みます。

My final advice is to embed async into existing systems. Just because you are not using an async-aware server like Falcon, doesn't mean you can't embed async into your controllers or workers to gain the benefits of concurrent execution.

```ruby
class FanOutWorker < ApplicationWorker
  def perform(urls)
    Sync do
      internet = Async::HTTP::Internet.instance

      tasks = urls.map do |url|
        task.async do
          response = internet.get(url)
          [url, response.read]
        ensure
          response.close
        end
      end

      responses = tasks.map(&:wait)
      # ... do something with responses ...
    end
  end
end
```

Here is an example of a worker which could be running on a job server such as Sidekiq, which isn't specifically aware of Async.

```ruby
class FanOutWorker < ApplicationWorker
  def perform(urls)
    Sync do
      internet = Async::HTTP::Internet.instance

      tasks = urls.map do |url|
        task.async do
          response = internet.get(url)
          [url, response.read]
        ensure
          response.close
        end
      end

      responses = tasks.map(&:wait)
      # ... do something with responses ...
    end
  end
end
```

It's the same fan-out code as before…

```ruby
class FanOutWorker < ApplicationWorker
  def perform(urls)
    Sync do
      internet = Async::HTTP::Internet.instance

      tasks = urls.map do |url|
        task.async do
          response = internet.get(url)
          [url, response.read]
        ensure
          response.close
        end
      end

      responses = tasks.map(&:wait)
      # ... do something with responses ...
    end
  end
end
```

but we embed it in a top level Sync block. This creates an event loop if required and runs the given block in it. This works equally well for Puma, Unicorn or even a command line script.

Now you have asynchronous HTTP in Ruby...

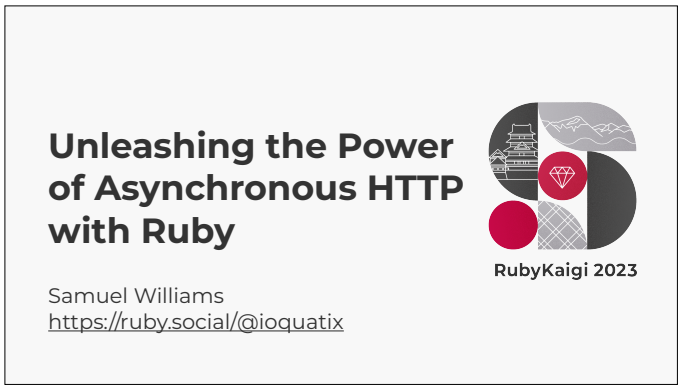
これで、Ruby で非同期 HTTP が使用できるようになりました。。。

So, now I have given you the tools to build asynchronous HTTP in Ruby.

Unleashing the Power of Asynchronous HTTP with Ruby

どうやって解き放つのか見せてください。

Please show me how you are going to unleash the power of asynchronous HTTP with Ruby. I look forward to seeing your results.



Thank you for coming to my presentation. If you have any questions, please feel free to contact me.