# Ruby - Feature #20215

## Introduce `IO#readable?`

01/26/2024 05:19 AM - ioquatix (Samuel Williams)

| | |
|---|---|
| **Status:** | Open |
| **Priority:** | Normal |
| **Assignee:** | |
| **Target version:** | |

**Description**

There are some cases where, as an optimisation, it's useful to know whether more data is potentially available.

We already have IO#eof? but the problem with using IO#eof? is that it can block indefinitely for sockets.

Therefore, code which uses IO#eof? to determine if there is potentially more data, may hang.

```
def make_request(path = "/")
  client = connect_remote_host
  # HTTP/1.0 request:
  client.write("GET #{path} HTTP/1.0\r\n\r\n")

  # Read response
  client.gets("\r\n") # => "HTTP/1.0 200 OK\r\n"

  # Assuming connection close, there are two things the server can do:
  # 1. peer.close
  # 2. peer.write(...); peer.close

  if client.eof? # <--- Can hang here!
    puts "Connection closed"
    # Avoid yielding as we know there definitely won't be any data.
  else
    puts "Connection open, data may be available..."
    # There might be data available, so yield.
    yield(client)
  end
ensure
  client&.close
end

make_request do |client|
  puts client.read # <--- Prefer to wait here.
end
```

The proposed IO#readable? is similar to IO#eof? but rather than blocking, would simply return false. The expectation is the user will subsequently call read which may then wait.

The proposed implementation would look something like this:

```
class IO
  def readable?
    !self.closed?
  end
end

class BasicSocket
  # Is it likely that the socket is still connected?
  # May return false positive, but won't return false negative.
  def readable?
    return false unless super

    # If we can wait for the socket to become readable, we know that the socket may still be open.
    result = self.recv_nonblock(1, MSG_PEEK, exception: false)
```

```
    # No data was available - newer Ruby can return nil instead of empty string:
    return false if result.nil?

    # Either there was some data available, or we can wait to see if there is data avaialble.
    return !result.empty? || result == :wait_readable

  rescue Errno::ECONNRESET
    # This might be thrown by recv_nonblock.
    return false
  end
end
```

For IO itself, when there is buffered data, readable? would also return true immediately, similar to eof?. This is not shown in the above implementation as I'm not sure if there is any Ruby method which exposes "there is buffered data".

### History

#### #1 - 01/26/2024 06:17 AM - ioquatix (Samuel Williams)

*- Description updated*

#### #2 - 02/01/2024 06:19 AM - shyouhei (Shyouhei Urabe)

After thinking it for a while, yes I'm for this feature.  There seems to be no way right now to achieve what is needed.

#### #3 - 02/01/2024 02:59 PM - Dan0042 (Daniel DeLorme)

I'm interested in this. I previously had to implement a "nonblocking gets" which was a bit tricky.
It would have been nice to just write io.gets if io.readable?

But I want to confirm this is the expected behavior: #readable? should return true if there is data immediately available for #read
Based on the BasicSocket implementation above, it seems like #readable? could return true even if no data is available?

#### #4 - 02/02/2024 09:06 PM - Eregon (Benoit Daloze)

Would io.wait_readable(0) work instead? If not, why not?

#### #5 - 02/03/2024 05:13 PM - Dan0042 (Daniel DeLorme)

Interesting, I didn't know about #wait_readable. Looks like previously it needed require "io/wait" but it was integrated in Ruby 3.2 core.

Unfortunately it doesn't work for my use case:

```
(echo 1;sleep 1;echo 2)|3.2 ruby -e '8.times{p(gets:$stdin.gets, closed:$stdin.closed?) if p $stdin.wait_reada
ble(0); sleep 0.2}'
#<IO:<STDIN>>
{:gets=>"1\n", :closed=>false}
nil
nil
nil
nil
#<IO:<STDIN>>
{:gets=>"2\n", :closed=>false}
#<IO:<STDIN>>
{:gets=>nil, :closed=>false}
#<IO:<STDIN>>
{:gets=>nil, :closed=>false}
```

If #wait_readable returns #<IO:<STDIN>>, the subsequent #gets should never return nil.

#### #6 - 02/03/2024 07:52 PM - Dan0042 (Daniel DeLorme)

Actually if I think about it a little, I can work around that like this:

```
loop do
  if $stdin.wait_readable(0)
    str = $stdin.gets or abort("<eof>")
    p str
  else
    puts "no input, let's wait a bit..."
    sleep 1
  end
end
```

And I now realize this is the same as IO.select; not sure why I didn't think of using that before.

Which brings me to understand what @ioquatix (Samuel Williams) has in mind is a bit different from what I thought. It looks like it can be implemented like this?

```
class IO
  def readable?
    if IO.select([self],[],[],0).nil?
      true #no data available but not eof
    else
      !eof? #will not hang due to check above
    end
  end
  #or, what about adding an argument to eof?
  def eof?(non_blocking=false)
    if non_blocking
      return nil if IO.select([self],[],[],0).nil?
    end
    super
  end
end
```

#### #7 - 02/19/2024 06:44 PM - forthoney (Seong-Heon Jung)

I think the name is potentially confusing. Consider the following:

```
if client.readable?
  client.read # this may block
end
```

I personally would be a bit surprised if client.read blocked despite client.readable? returning true. readable is ambiguous between "there is something to read right now" and "there will eventually be something to read".

#### #8 - 04/16/2024 01:13 AM - ioquatix (Samuel Williams)

> Would io.wait_readable(0) work instead? If not, why not?

(1) I don't think it makes sense to add wait_* to StringIO but it does make sense to add readable? to StringIO.
(2) wait_readable(0) is impossible to replicate the desired behaviour:

```
> r, w = Socket.pair(:UNIX, :STREAM)
=> [#<Socket:fd 5>, #<Socket:fd 6>]
irb(main):003> r.wait_readable(0)
=> nil # Not readable (no data available right now within the timeout specified)
irb(main):004> r.readable?
=> true # It's possible to read from this socket.
irb(main):005> w.close
=> nil
irb(main):006> r.wait_readable(0)
=> #<Socket:fd 5> # It's possible to read the "close" (zero-size).
irb(main):007> r.readable?
=> false # The socket is not readable (call to `#read` will not give data).
```

After thinking about it for a while, IO#readable? is like the inverse of IO#eof? without any blocking. An alternative proposal might be to make IO#eof? non-blocking which is probably less risky than the current implementation of eof? but might be incompatible with current usage.

#### #9 - 04/16/2024 09:54 AM - ioquatix (Samuel Williams)

After some discussion, I investigated libc's feof:

1. https://github.com/bminor/glibc/blob/14e56bd4ce15ac2d1cc43f762eb2e6b83fec1afe/libio/feof.c
2. https://github.com/bminor/glibc/blob/14e56bd4ce15ac2d1cc43f762eb2e6b83fec1afe/libio/bits/types/struct_FILE.h#L112

It looks like the expected behaviour of "eof?" should be non-blocking.

#### #10 - 04/16/2024 05:36 PM - mame (Yusuke Endoh)

Before the dev meeting, I talked with @akr (Akira Tanaka), who designed the Ruby IO, but we could not understand the motivation of the following branch.

```
if client.eof? # <--- Can hang here!
```

Can you explain what you want to do by this branch? Even if client.eof? returns false without blocking, it could still result in an EOF with zero read bytes. Therefore, it would be better to do read without unnecessary checks.

#### #11 - 04/17/2024 12:07 AM - ioquatix (Samuel Williams)

> Even if client.eof? returns false without blocking, it could still result in an EOF with zero read bytes. Therefore, it would be better to do read without unnecessary checks.

I understand, thanks for your question. I may not be able to answer this well before the meeting, so I'll try to come up with a clear justification, but if the time frame is too tight, I'll discuss it next time instead.

#### #12 - 04/17/2024 01:13 AM - akr (Akira Tanaka)

I couldn't understand the motivation of this issue.

However, the state of the read side of unidirectional data flow (Unix pipe, half of a stream socket, etc.) can be one of the following.

1. We can read 1 or more bytes immediately.
2. We can detect EOF immediately.
3. We cannot determine data/EOF immediately. If we wait indefinitely, we can read 1 or more bytes.
4. We cannot determine data/EOF immediately. If we wait indefinitely, we can detect EOF.

io.eof? returns true on 2 and 4, false otherwise. (So, it blocks on 3 and 4.)
io.wait_readable(0) returns truthy on 1 and 2, falsy otherwise.

They are enough to distinguish the states.
If we don't want to block, it is impossible to distinguish 3 and 4, though.

#### #13 - 04/17/2024 04:25 AM - ioquatix (Samuel Williams)

After considering the various use cases I have, I think the easiest to describe problem is knowing whether a connection is still "connected" or not, i.e. whether read will definitely fail or might succeed.

I added a full working example of the problem here: https://github.com/socketry/protocol-http1/blob/540551bdbdbca06d746b4c4545af2d73ebcc7dcc/examples/http1/client.rb#L70. You can try different implementation of IO#readable? to see the behaviour.

The example demonstrates HTTP/1 persistent connection handling, where the remote server may at any time disconnect. In server.rb, it has a 50% chance of disconnecting. client.rb makes 10 connections, and tries to use persistent connections.

The key problem that I'm trying to address, is that there is no protocol-level mechanism to advertise that the remote server is closing the connection (in contrast, HTTP/2 has such a feature). So, what that means, is in the request loop, when we want to write the request, we want to ensure, with the best effort possible, that the connection is still alive and working. That is the purpose of IO#readable? in this context - whether there is a significantly good chance that writing an HTTP request will be successful.

In practice, persistent connections may sit in a connection pool for minutes or hours, and thus when you come to write a request, there is no easy operation to check "Is this connection still working?". That is the purpose of IO#readable?. Specifically, before writing a request, we check if the connection is still readable.

The logic for "Is the connection still readable?" depends on the situation and the underlying IO. As you know there are many different semantics for handling Sockets, Pipes, and so on, and we even provide our own blended semantics in StringIO. I'd like to introduce IO#readable?, BasicSocket#readable? based on recv_nonblock and StringIO#readable? which is similar to non-blocking eof?.

In other words, in the case of sockets, BasicSocket#readable? is querying the operating system to find out if the TCP connection is still working (i.e. not closed explicitly).

It's true that this can be a race condition, for example the TCP reset/shutdown could be delayed or received while writing the request. However, it's still better to prevent writing the request entirely if possible. That's because not all requests are idempotent e.g. POST requests for handling payments. It's much better to know ahead of time that the request will fail because the persistent connection has been shut down, than to find out half way through writing the non-idempotent request.

Example output from the client:

```
> bundle exec client.rb
Connected to #<Socket:0x0000754fc63d8b60> #<Addrinfo: 127.0.0.1:8080 TCP>
Writing request...
Reading response...
Got response: ["HTTP/1.1", 200, "OK", #<Protocol::HTTP::Headers [["Content-Type", "text/plain"]]>, #<Protocol:
:HTTP1::Body::Fixed length=11 remaining=11>]
Hello World
Writing request...
Reading response...
```

```
Got response: ["HTTP/1.1", 200, "OK", #<Protocol::HTTP::Headers [["Content-Type", "text/plain"]]>, #<Protocol:
:HTTP1::Body::Fixed length=11 remaining=11>]
Hello World
Writing request...
Reading response...
Got response: ["HTTP/1.1", 200, "OK", #<Protocol::HTTP::Headers [["Content-Type", "text/plain"]]>, #<Protocol:
:HTTP1::Body::Fixed length=11 remaining=11>]
Hello World
Writing request...
Reading response...
Got response: ["HTTP/1.1", 200, "OK", #<Protocol::HTTP::Headers [["Content-Type", "text/plain"]]>, #<Protocol:
:HTTP1::Body::Fixed length=11 remaining=11>]
Hello World
Writing request...
Client is not readable, closing...
Reconnecting...
Connected to #<Socket:0x0000754fc63d0fa0> #<Addrinfo: 127.0.0.1:8080 TCP>
Reading response...
Got response: ["HTTP/1.1", 200, "OK", #<Protocol::HTTP::Headers [["Content-Type", "text/plain"]]>, #<Protocol:
:HTTP1::Body::Fixed length=11 remaining=11>]
Hello World
Writing request...
Client is not readable, closing...
Reconnecting...
Connected to #<Socket:0x0000754fc644d2d0> #<Addrinfo: 127.0.0.1:8080 TCP>
Reading response...
Got response: ["HTTP/1.1", 200, "OK", #<Protocol::HTTP::Headers [["Content-Type", "text/plain"]]>, #<Protocol:
:HTTP1::Body::Fixed length=11 remaining=11>]
Hello World
Writing request...
Client is not readable, closing...
Reconnecting...
Connected to #<Socket:0x0000754fc6448cf8> #<Addrinfo: 127.0.0.1:8080 TCP>
Reading response...
Got response: ["HTTP/1.1", 200, "OK", #<Protocol::HTTP::Headers [["Content-Type", "text/plain"]]>, #<Protocol:
:HTTP1::Body::Fixed length=11 remaining=11>]
Hello World
Writing request...
Client is not readable, closing...
Reconnecting...
Connected to #<Socket:0x0000754fc6443f00> #<Addrinfo: 127.0.0.1:8080 TCP>
Reading response...
Got response: ["HTTP/1.1", 200, "OK", #<Protocol::HTTP::Headers [["Content-Type", "text/plain"]]>, #<Protocol:
:HTTP1::Body::Fixed length=11 remaining=11>]
Hello World
Writing request...
Client is not readable, closing...
Reconnecting...
Connected to #<Socket:0x0000754fc64bfe20> #<Addrinfo: 127.0.0.1:8080 TCP>
Reading response...
Got response: ["HTTP/1.1", 200, "OK", #<Protocol::HTTP::Headers [["Content-Type", "text/plain"]]>, #<Protocol:
:HTTP1::Body::Fixed length=11 remaining=11>]
Hello World
Writing request...
Client is not readable, closing...
Reconnecting...
Connected to #<Socket:0x0000754fc64bc2e8> #<Addrinfo: 127.0.0.1:8080 TCP>
Reading response...
Got response: ["HTTP/1.1", 200, "OK", #<Protocol::HTTP::Headers [["Content-Type", "text/plain"]]>, #<Protocol:
:HTTP1::Body::Fixed length=11 remaining=11>]
Hello World
Closing client...
Exiting.
```

Note that Client is not readable, closing... indicates that the client was closed before the request was written, which is the ideal case.

#### #14 - 04/17/2024 11:16 PM - akr (Akira Tanaka)

Thank you for the explanation of the motivation.
I feel it is reasonable.

However, mame-san still had a question yesterday: why don't you detect an error in writing a request?
I guess it is because writing the request may not fail.
It needs two writes to detect the error.
(The first write in a client application causes the client kernel to send a packet to the server.

The server kernel responds to it with a RST packet because the server socket is closed.
The client kernel receives the RST packet and detects we cannot send data in the connection.
The second write in the client application will cause an error.)
Please point out if I am wrong.

My next question is why io.eof? and io.wait_readable(0) doesn't fulfill your motivation.
My current guess is related to the buffering mechanism of IO class.
But I'm not certain.

#### #15 - 04/19/2024 01:20 AM - Dan0042 (Daniel DeLorme)

ioquatix (Samuel Williams) wrote in #note-13:

> In practice, persistent connections may sit in a connection pool for minutes or hours, and thus when you come to write a request, there is no easy operation to check "Is this connection still working?". That is the purpose of IO#readable?.
> In other words, in the case of sockets, BasicSocket#readable? is querying the operating system to find out if the TCP connection is still working (i.e. not closed explicitly).

That makes a lof of sense to me, from personal experience. But I implore you to reconsider the naming readable?
Just like @forthoney, I personally would be quite surprised if client.read blocked despite client.readable? returning true. If the purpose is to check that the connection is still open, then maybe #still_open? would work as a name? Actually, given the description above that mentions "if the TCP connection is still working", I'm not quite sure why you say this method is like eof? rather than closed?

#### #16 - 04/22/2024 04:06 AM - ioquatix (Samuel Williams)

Regarding naming, I had other ideas like open? or connected? but I think they have their own issues. I think my preference is readable? but I'd like to hear alternatives. Maybe open? is okay but it sounds like the opposite of closed? which strictly speaking it isn't.

Regarding implementation, I spent several hours debugging usage of eof?. Because eof? can mutate the underlying buffer, it's not concurrency safe either. This was the source of an extremely hard to diagnose bug where blocking read and eof? occurred in different fibers.. Maybe we should create a separate issue, to make eof? safer (i.e. it doesn't call io_fillbuf internally).

#### #17 - 04/22/2024 04:08 AM - ioquatix (Samuel Williams)

> I'm not quite sure why you say this method is like eof? rather than closed?

We work with the interface and taxonomy given to us by POSIX.

close(fd) causes the file descriptor to become invalid.

It's different from what happens if the **remote end** closes or shuts down the connection. In that case, the file descriptor is still valid, but operations like read and write will fail.