



 Kaiqi on Rails 2025

避けられない I/O 待ちに対処する： Rails アプリにおける SSE と async gem の活用

Tackling Inevitable I/O Latency in Rails Apps with SSE and the async gem



Partner with RubyStackNews

Independent Ruby & Rails publication for senior developers

Why RubyStackNews?

- Focused on Ruby and Ruby on Rails
- Long-form articles based on real conference talks
- Audience of senior developers and tech leads
- Readers from the US, Europe, and Asia

RubyStackNews turns conference talks and real-world experience into practical, production-focused technical articles.

Partnerships & Sponsorships

- Article sponsorships
- Inline placements inside articles
- Sidebar visibility

[View partnership details](#)



Taiki Kawakami @moznion

SmartBank, Inc.

Software Engineer





頑張らなくていいお金の管理

— AI家計簿アプリ —

ワンバンク



SSE: server-sent events

ざっくり言うと

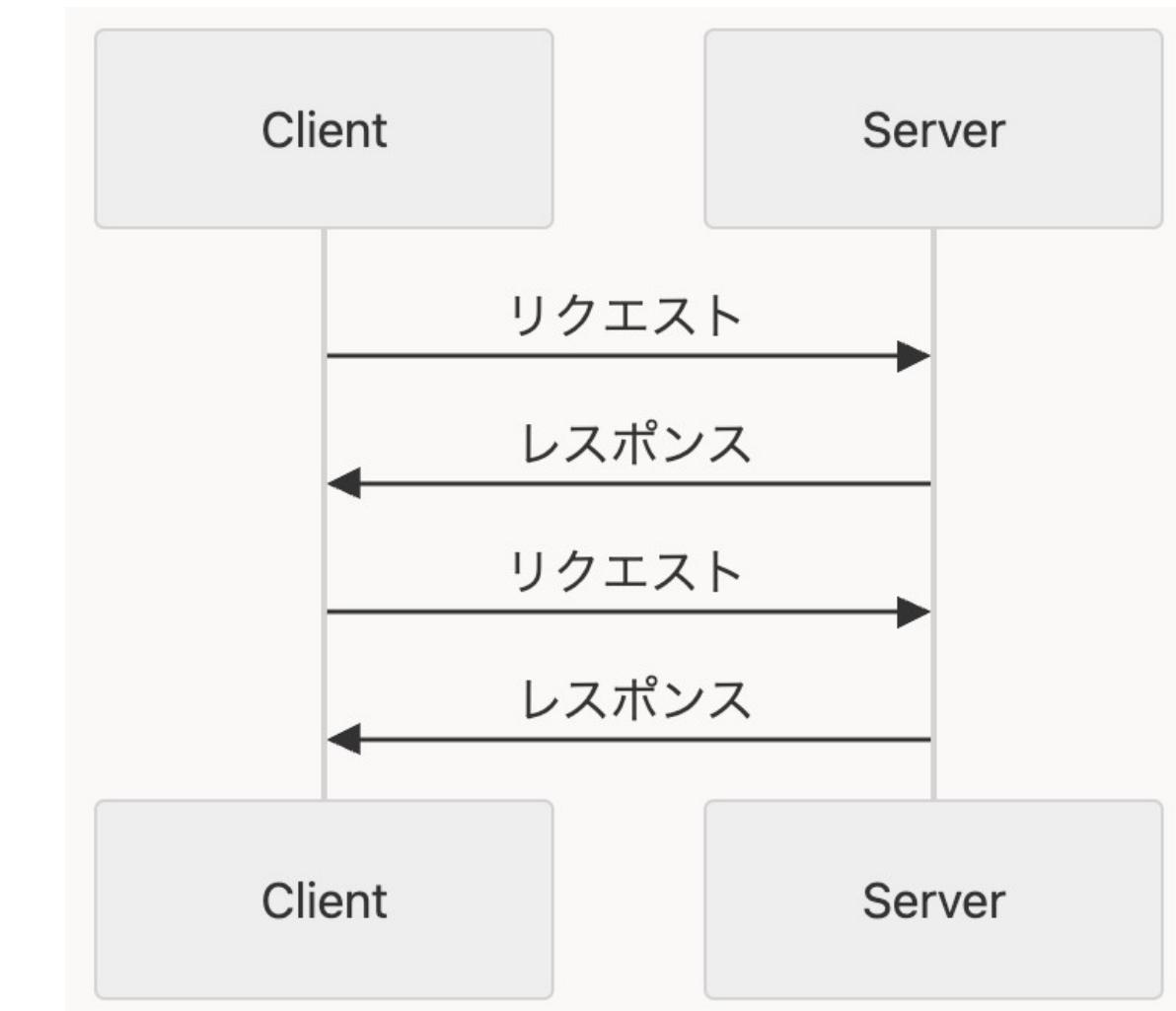
一度クライアント - サーバー間で
HTTP の SSE コネクションを張ると、
サーバーからクライアントに対して
継続的にデータを送出できる仕組み

”

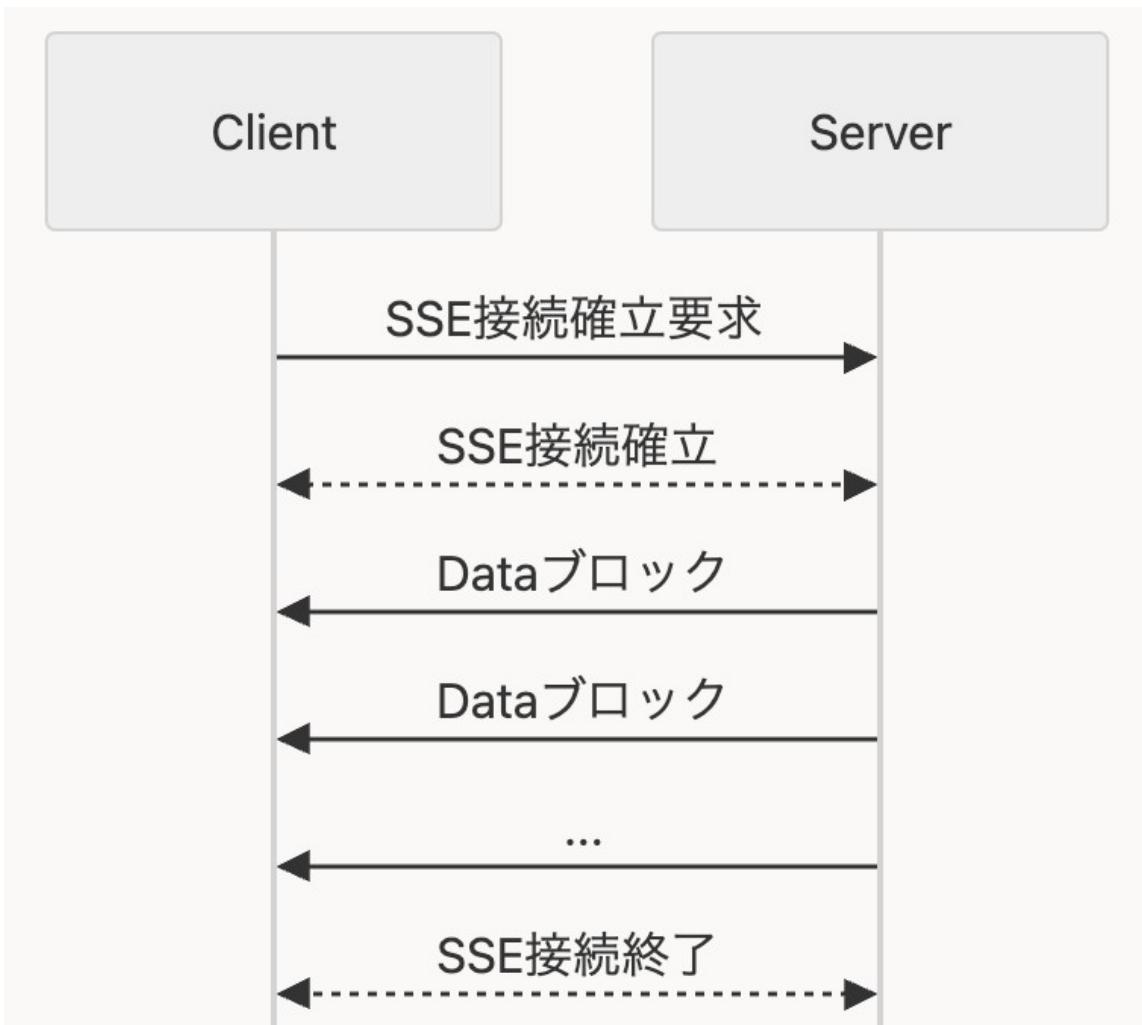
伝統的には、ウェブページが新たなデータを受け取るために、
サーバーにリクエストを送信しなければなりません。すなわち、
ページがサーバーからデータを要求します。サーバー送信イベン
トによって、サーバーがウェブページにメッセージをプッシュ送
信することにより、サーバーからウェブページへ新たなデータを
いつでも送信することができます。

<https://developer.mozilla.org/ja/docs/Web/API/Server-sentevents>

通常の HTTP



SSE



例：

- ▶ ニュースのアップデート（速報）
- ▶ ログのストリーミング表示
- ▶ メッセージング・チャット
- ▶ KoR のサブスクリーン†
- ▶ 長時間かかる処理の…
- ▶ 進捗表示
- ▶ 完了通知

ChatGPT 5

Does the ChatGPT app use Server-Sent Events (SSE) for interaction?

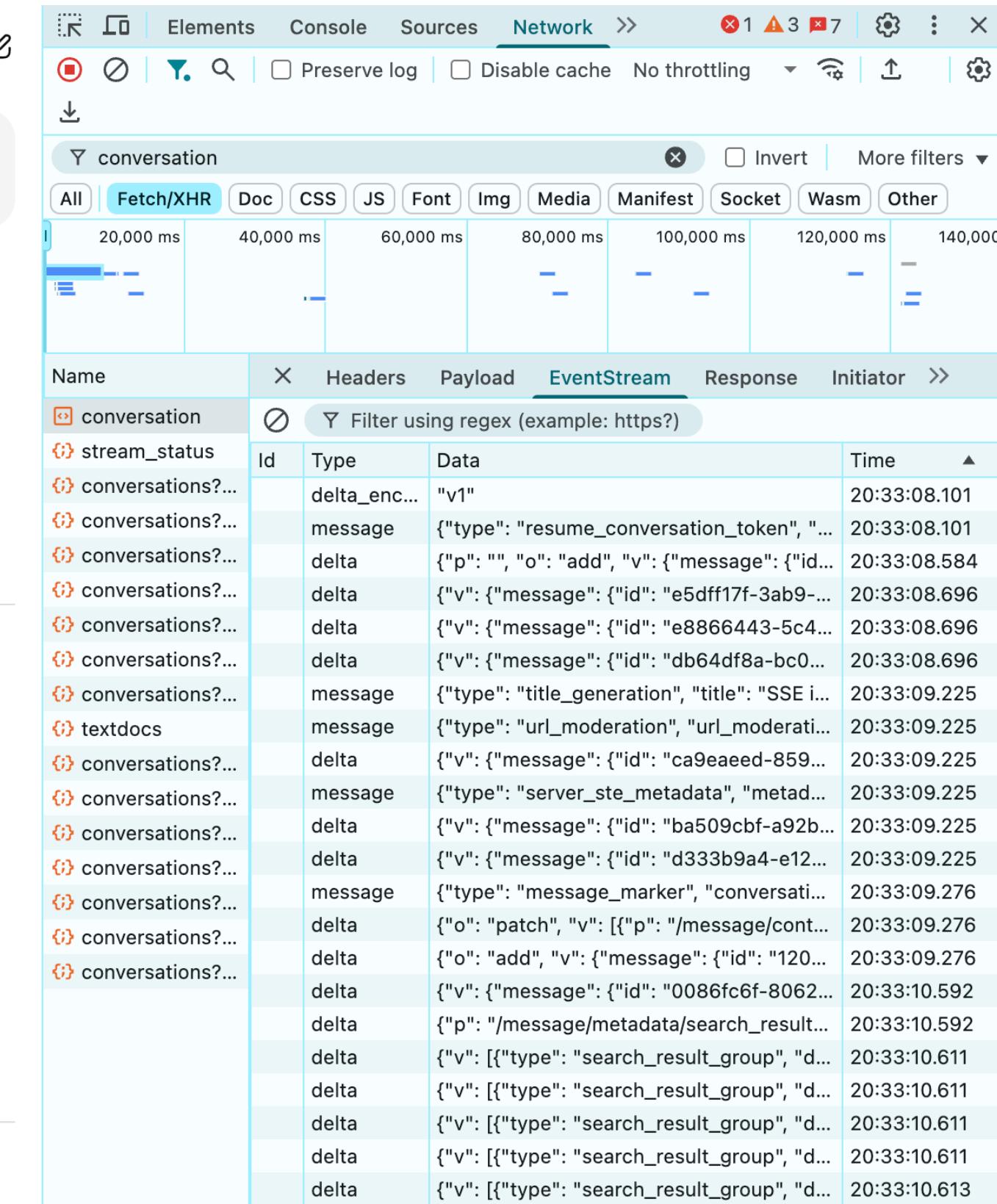
I couldn't find any *definitive public confirmation* from OpenAI that says "the ChatGPT native app uses SSE (Server-Sent Events)" as its streaming protocol. There are several sources (blogs, reverse-engineering, developer communities) that **claim** the web interface of ChatGPT uses SSE for streaming responses.

Here's what is known / plausible, and what remains uncertain:

What the public sources suggest

- A blog by Theodor's Blog says that the web ChatGPT front-end uses SSE: when you POST to the `/conversation` endpoint, the server responds via an "EventStream" (i.e. SSE) rather than a WebSocket.
- Other articles analyzing how ChatGPT streams text say that SSE is a simpler, adequate solution for one-way streaming (server→client) of incremental "tokens" or "deltas", which is exactly what is needed when generating text.
- The SSE approach has pros: less overhead than WebSockets when you don't need full bidirectional streaming, simpler to implement, works well over HTTP.

What is not confirmed / areas of uncertainty



↑ChatGPT も SSE で動作しているという例

例：

- ▶ ニュースのアップデート（速報）
- ▶ ログのストリーミング表示
- ▶ メッセージング・チャット
- ▶ KoR のサブスクリーン†
- ▶ 長時間かかる処理の…
 - ▶ 進捗表示
 - ▶ 完了通知

ChatGPT 5

Does the ChatGPT app use Server-Sent Events (SSE) for interaction?

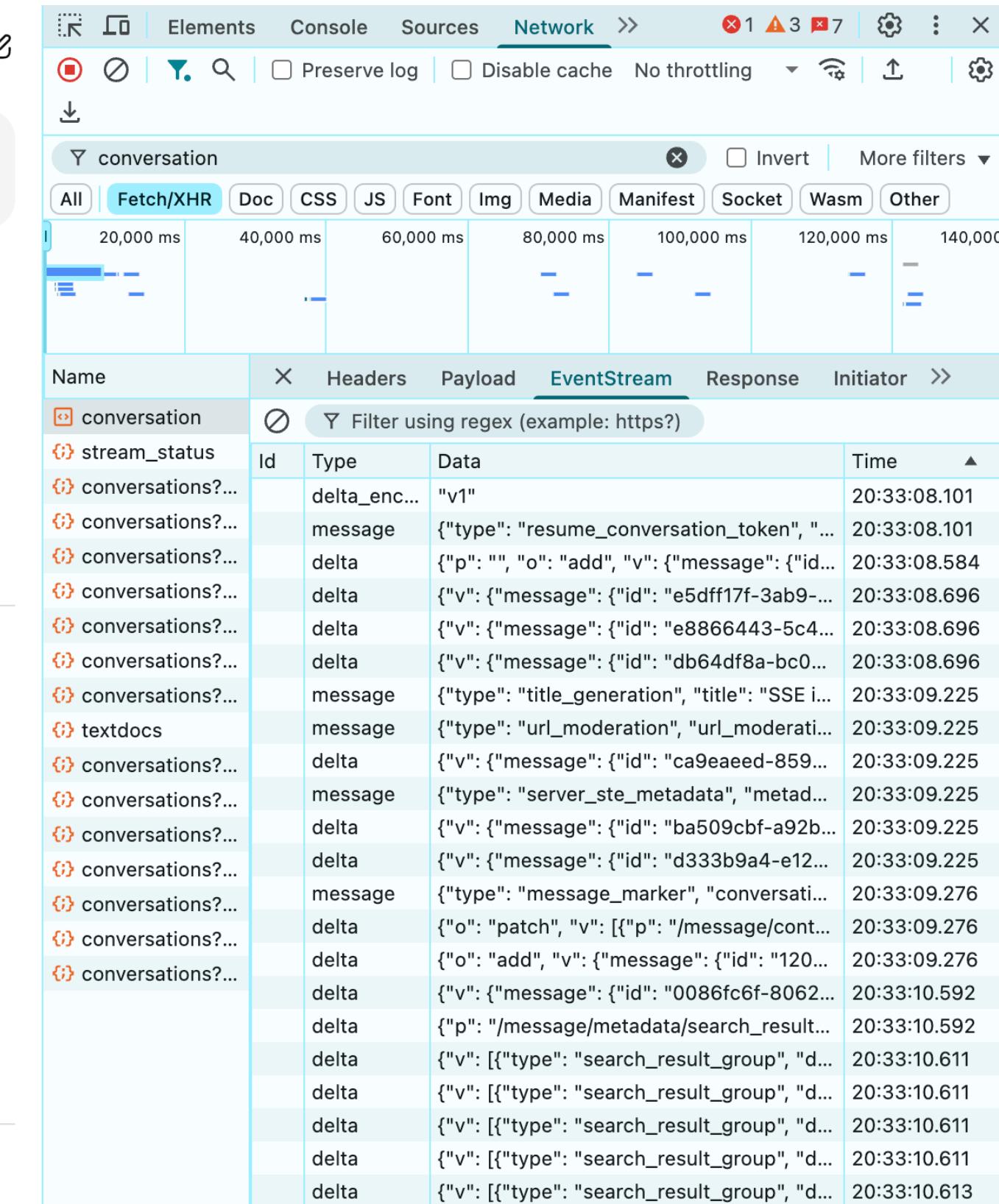
I couldn't find any *definitive public confirmation* from OpenAI that says "the ChatGPT native app uses SSE (Server-Sent Events)" as its streaming protocol. There are several sources (blogs, reverse-engineering, developer communities) that **claim** the web interface of ChatGPT uses SSE for streaming responses.

Here's what is known / plausible, and what remains uncertain:

What the public sources suggest

- A blog by Theodor's Blog says that the web ChatGPT front-end uses SSE: when you POST to the `/conversation` endpoint, the server responds via an "EventStream" (i.e. SSE) rather than a WebSocket.
- Other articles analyzing how ChatGPT streams text say that SSE is a simpler, adequate solution for one-way streaming (server→client) of incremental "tokens" or "deltas", which is exactly what is needed when generating text.
- The SSE approach has pros: less overhead than WebSockets when you don't need full bidirectional streaming, simpler to implement, works well over HTTP.

What is not confirmed / areas of uncertainty



↑ChatGPT も SSE で動作しているという例

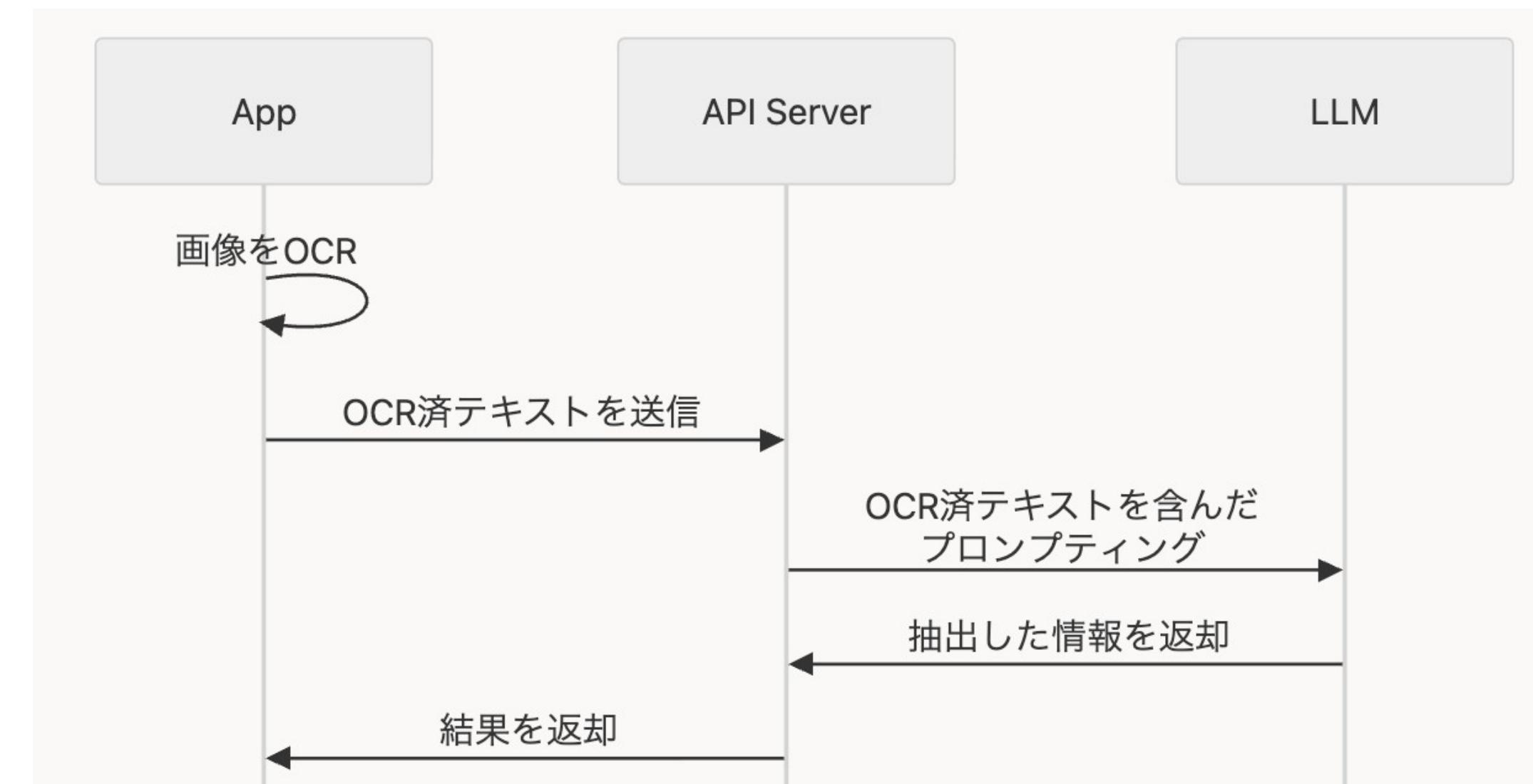
SSE ⇒ ユーザー体験向上施策の一種

例：長い待ち時間 (e.g., I/O wait) が発生する処理の進捗

昨今のアプリだとよくある

- ▶ API/RPC 呼び出し
- ▶ AI (LLM) 推論

右図はワンバンクアプリの「AI スクショ読み取り機能」



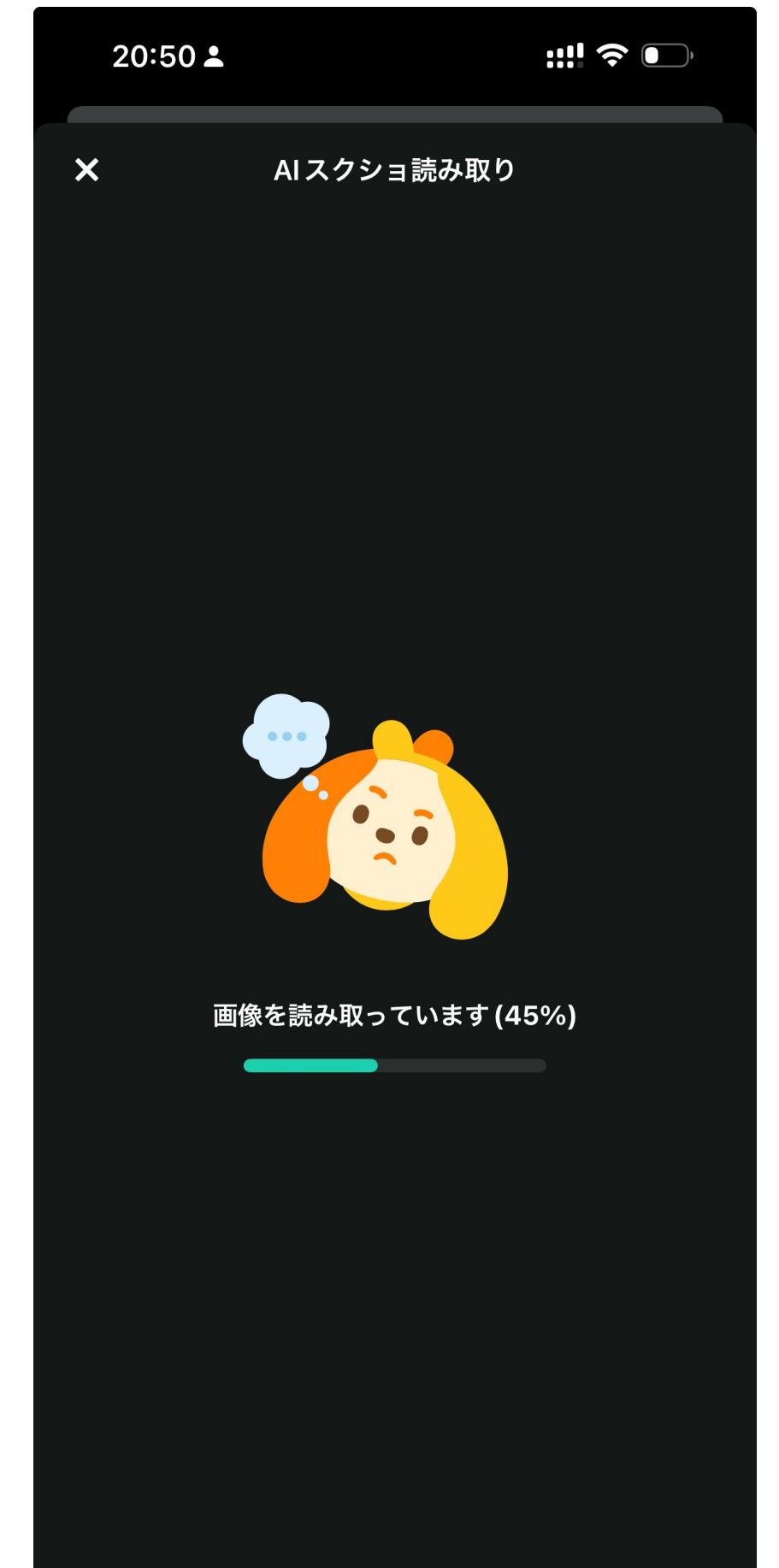
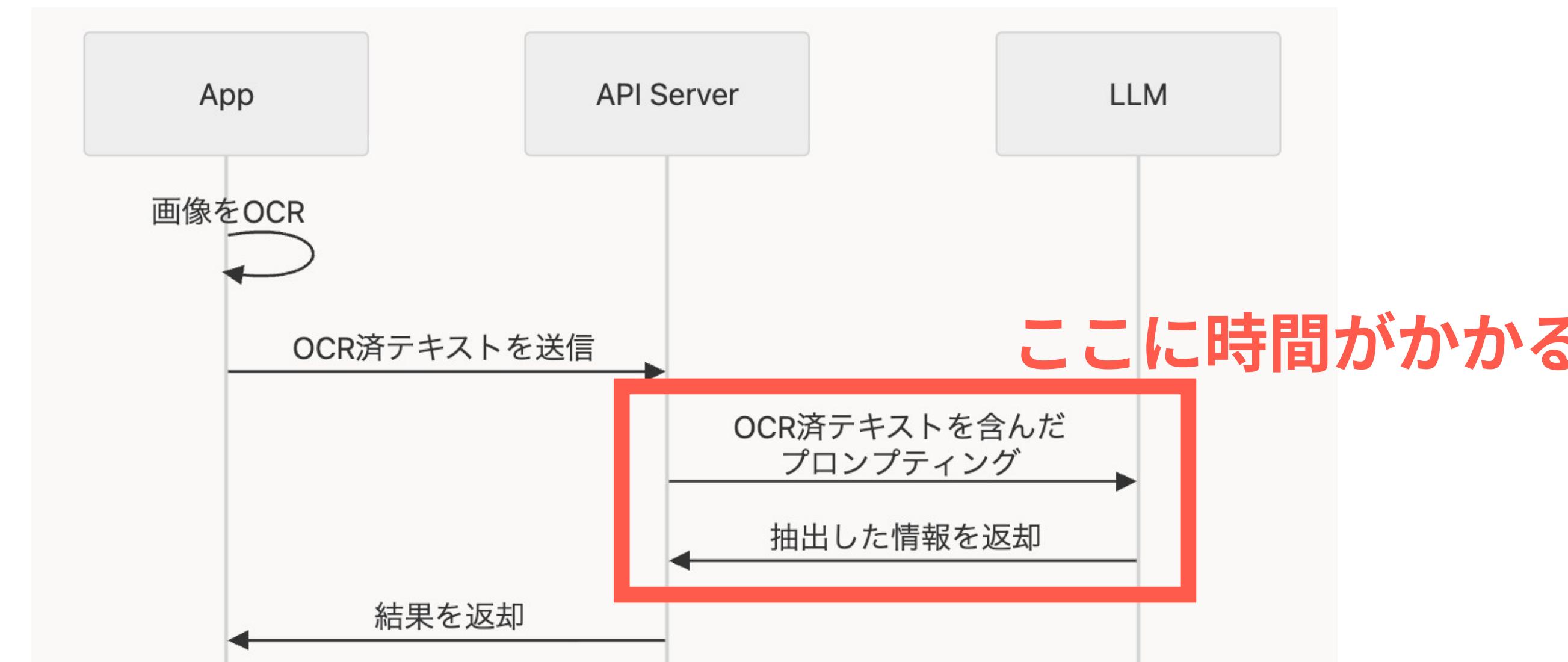
スクショ画像を OCR して LLM が情報抽出する機能

例：長い待ち時間 (e.g., I/O wait) が発生する処理の進捗

昨今のアプリだとよくある

- ▶ API/RPC 呼び出し
- ▶ AI (LLM) 推論

右図はワンバンクアプリの「AI スクショ読み取り機能」

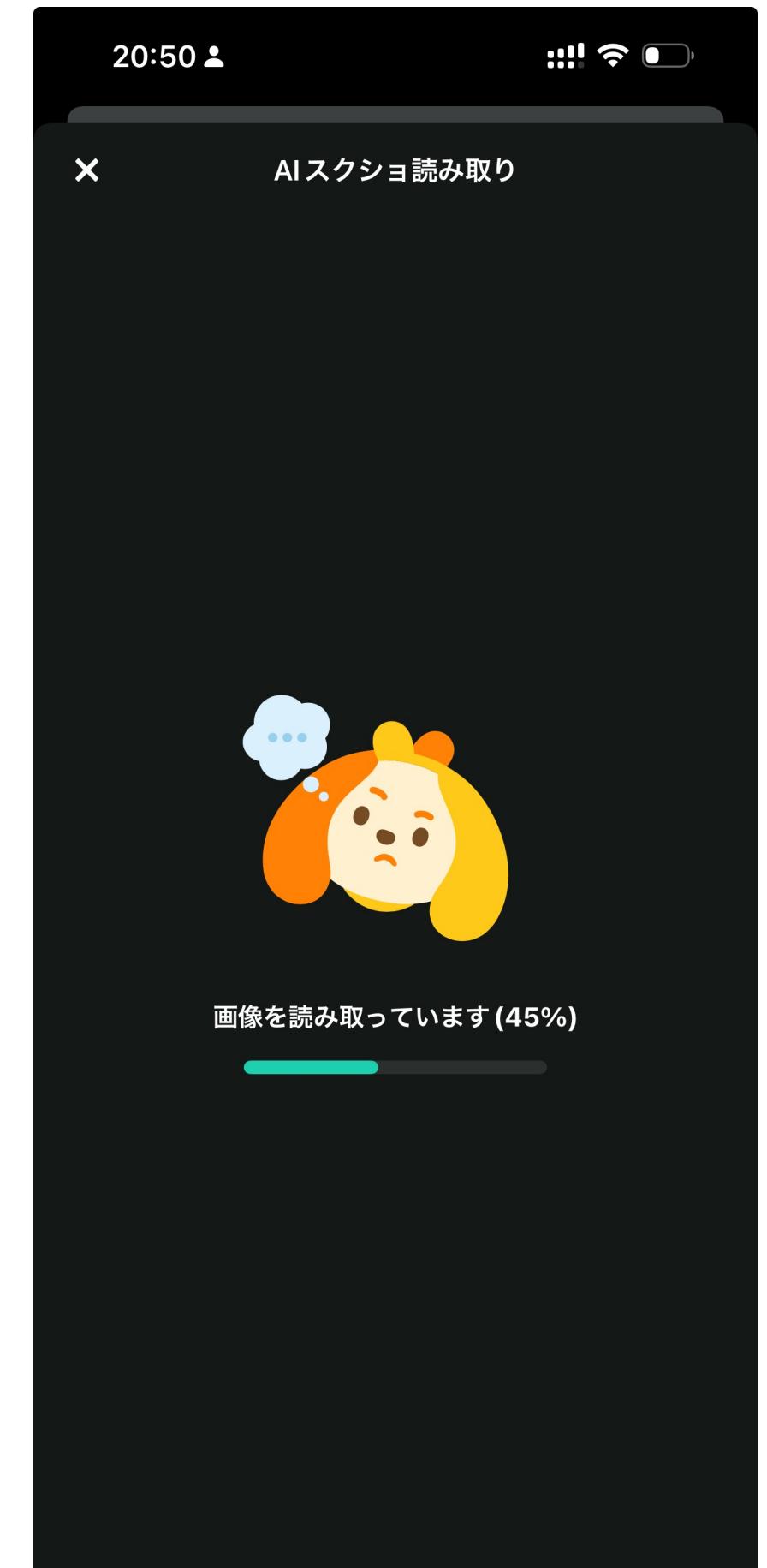
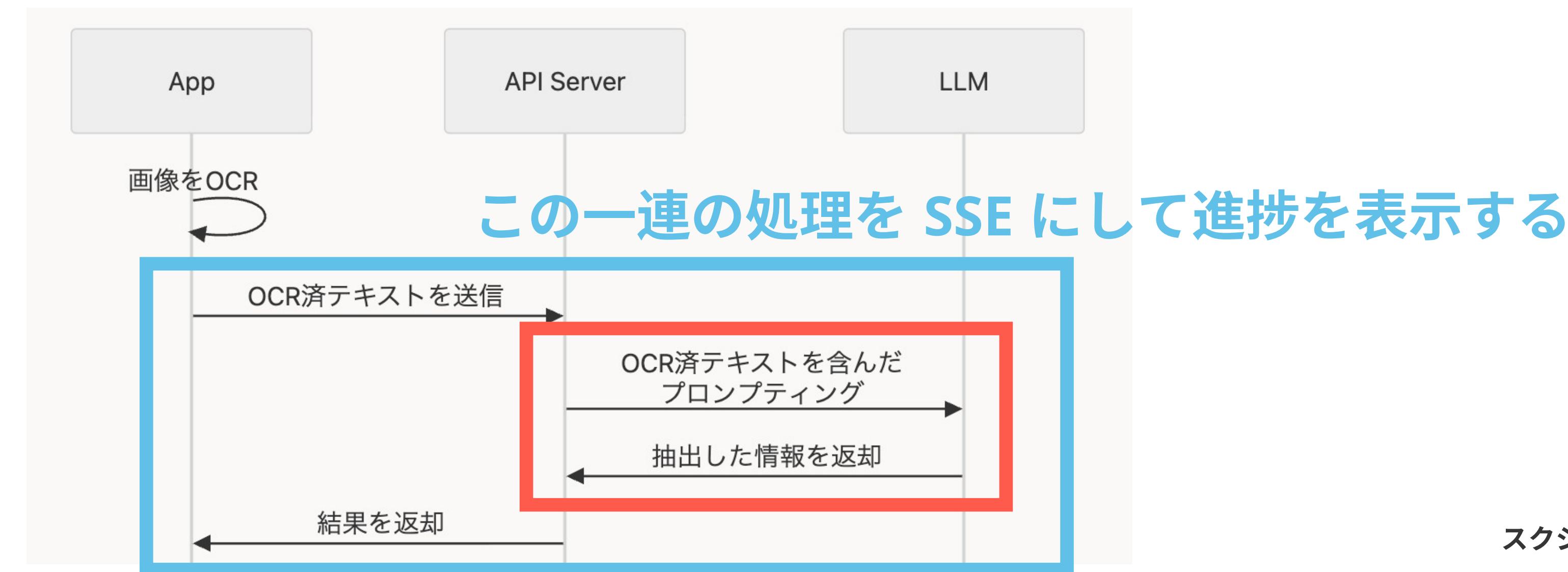


例：長い待ち時間 (e.g., I/O wait) が発生する処理の進捗

昨今のアプリだとよくある

- ▶ API/RPC 呼び出し
- ▶ AI (LLM) 推論

右図はワンバンクアプリの「AI スクショ読み取り機能」



スクショ画像を OCR して LLM が情報抽出する機能

例：長い待ち時間 (e.g., I/O wait) が発生する処理の進捗

- ▶ このような「待ち」に関する最適化の余地は限定されている
 - ▶ 「呼出先」の性能を良くすることは困難な場合が多い
- ▶ しかしユーザー体験は良くしたい
 - ▶ SSE はそのための選択肢のひとつになりえる

Rails で SSE Response を返却するには?

Example:

```
class MyController < ActionController::Base
  include ActionController::Live

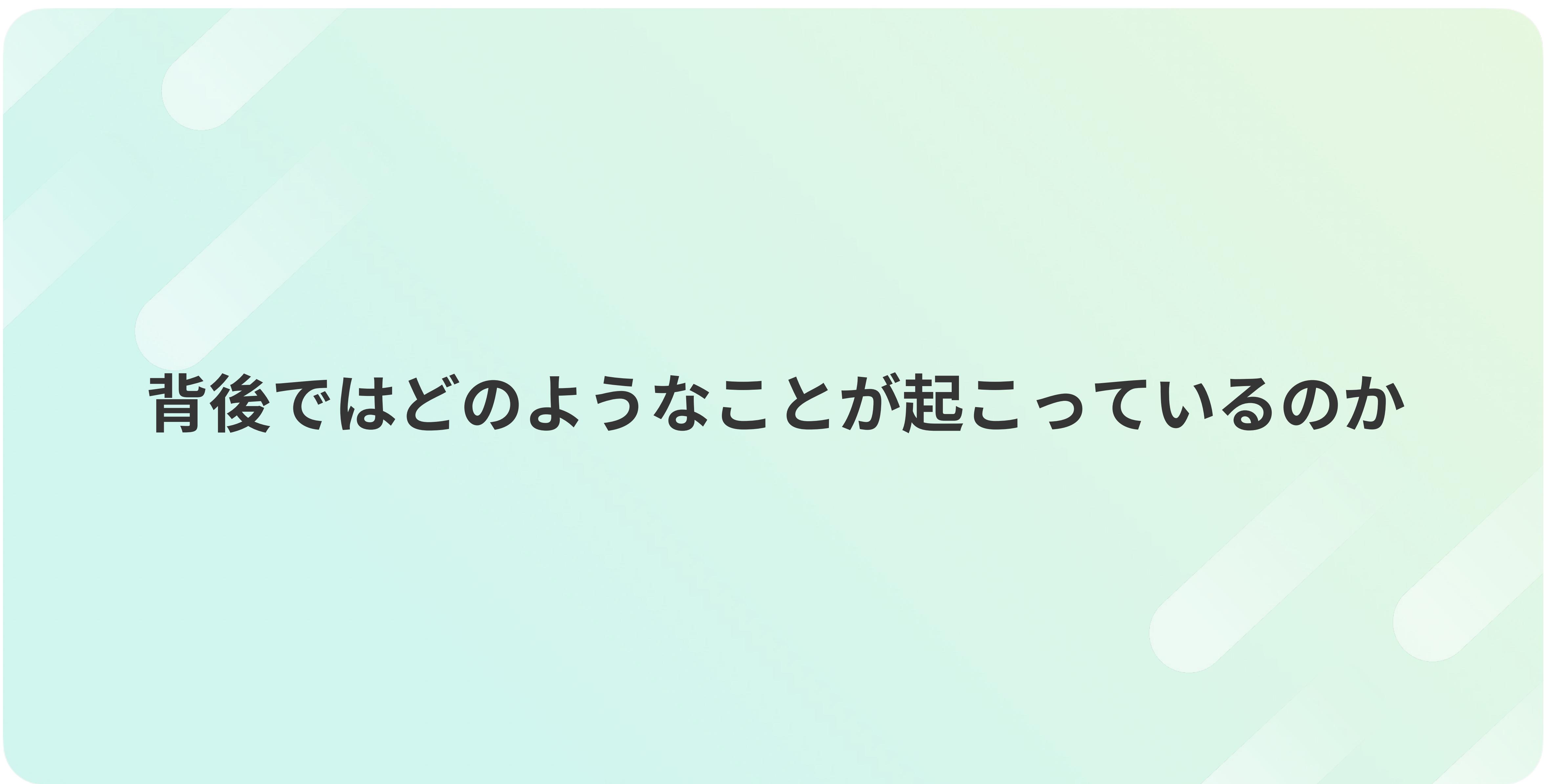
  def index
    response.headers['Content-Type'] = 'text/event-stream'
    sse = SSE.new(response.stream, retry: 300, event: "event-name")
    sse.write({ name: 'John' })
    sse.write({ name: 'John' }, id: 10)
    sse.write({ name: 'John' }, id: 10, event: "other-event")
    sse.write({ name: 'John' }, id: 10, event: "other-event", retry: 500)
  ensure
    sse.close
  end
end
```

出展 : <https://api.rubyonrails.org/classes/ActionController/Live/SSE.html>

簡単ですね



Kaiqi on Rails 2025



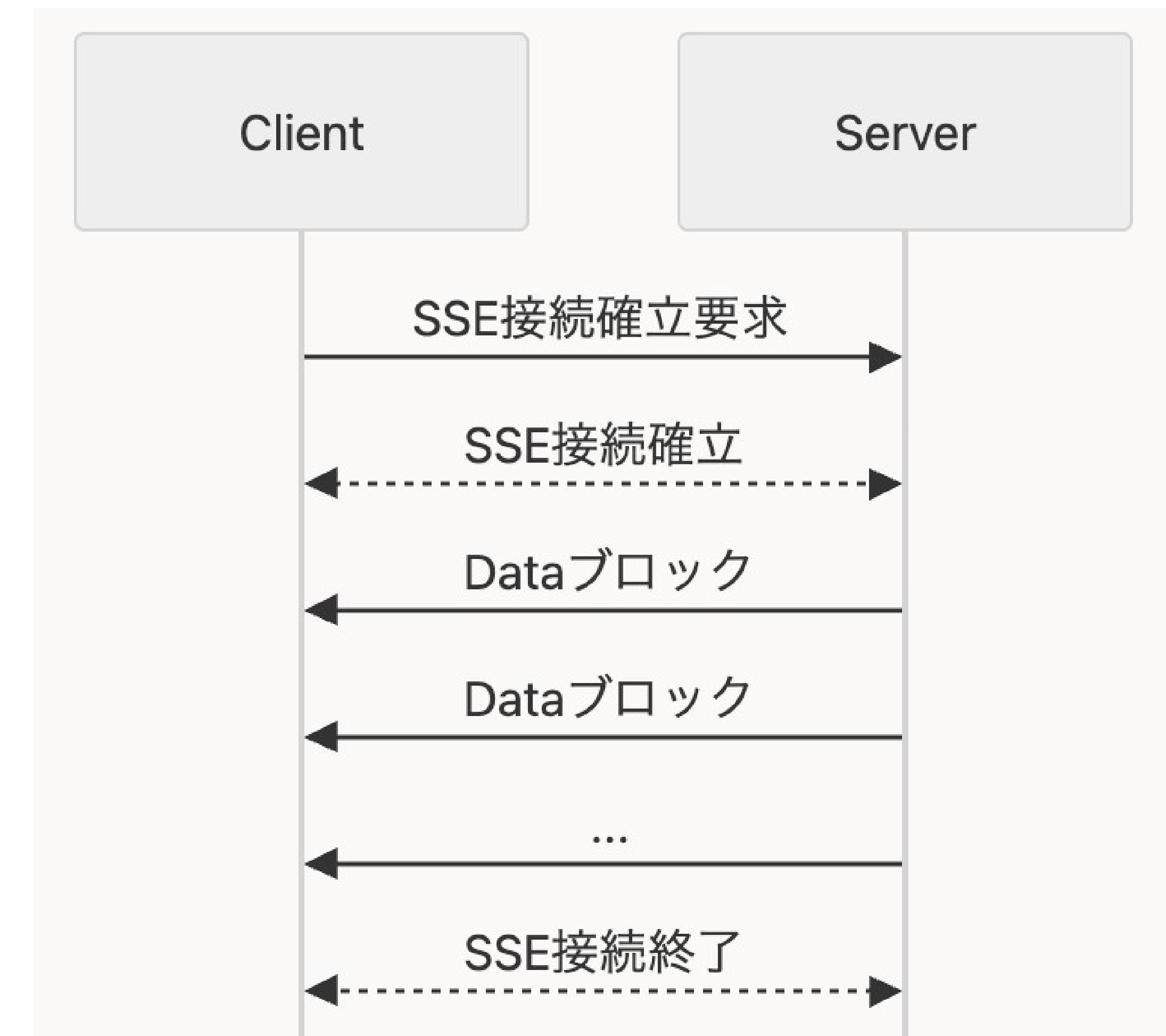
背後ではどのようなことが起こっているのか

以下が標準仕様 

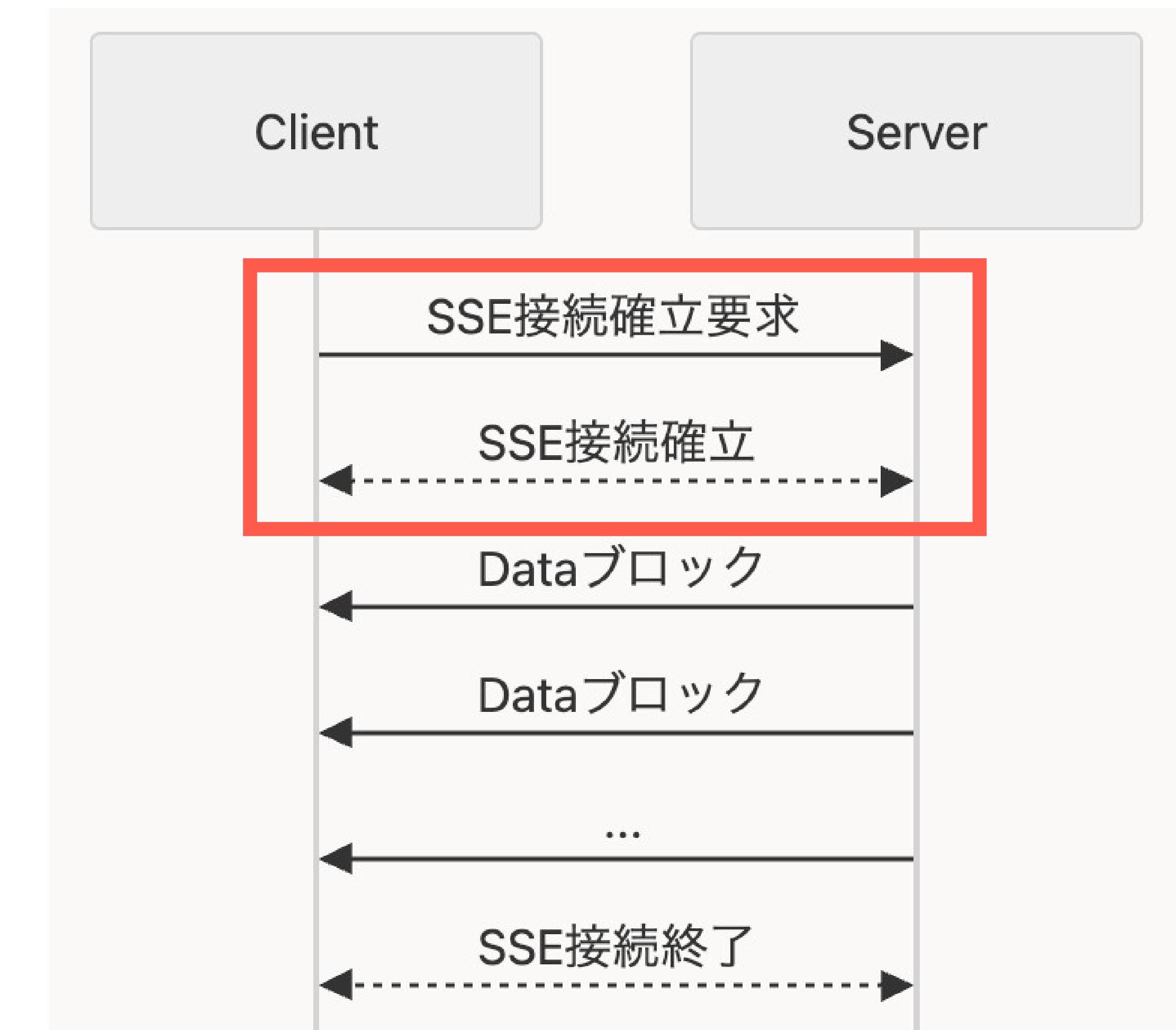
<https://html.spec.whatwg.org/multipage/server-sent-events.html>

- ▶ 基本的に HTTP 1.1 をベースとした”メッセージングフォーマット”
- ▶ WebSocket と比較してシンプルな仕組みになっている
- ▶ WebSocket によるもろもろの課題が HTTP 1.1 をベースにすることで解決されている
- ▶ 一方 SSE はサーバーからデータが一方的にプッシュされてくる仕組み
WebSocket のような 1 コネクションでの双方向性は無い

SSE プロトコルことはじめ



SSE 接続確立フェイズ



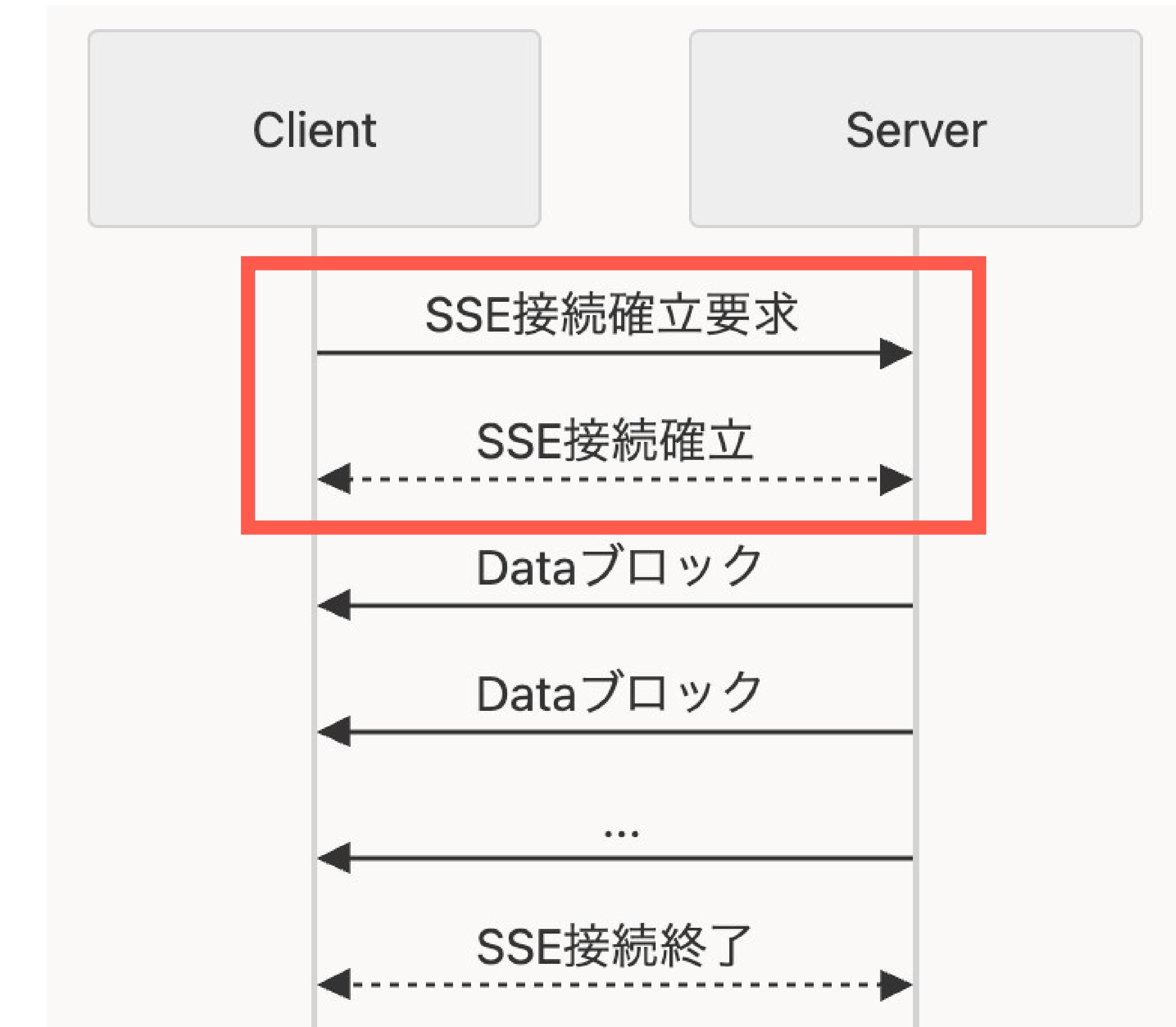
SSE 接続確立フェイズ

- ▶ Client: SSE のエンドポイントへ HTTP リクエストする
- ▶ Client ライブラリは内部的に SSE 接続のステート管理を行う

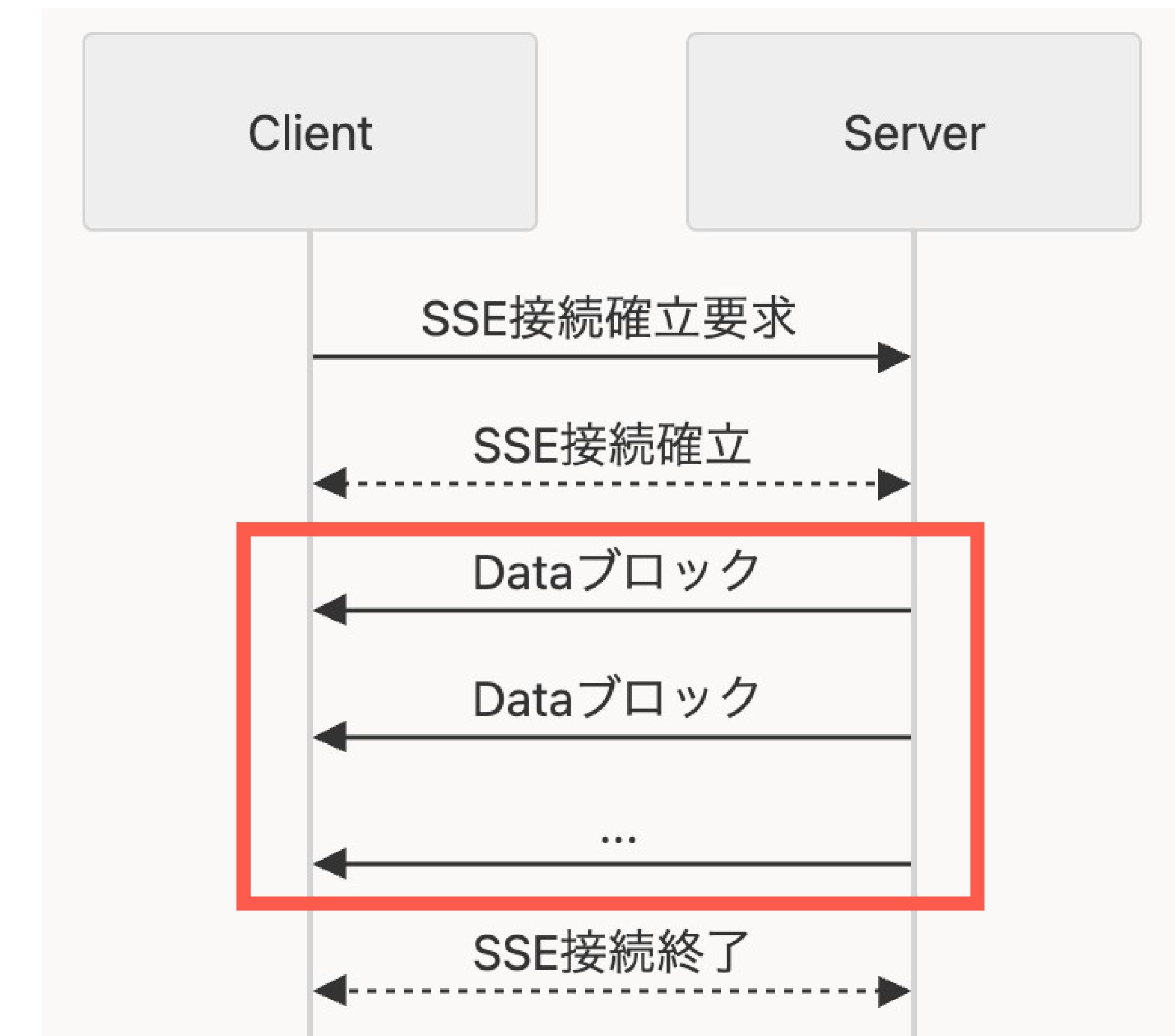
```
var source = new EventSource('updates.cgi');
source.onmessage = function (event) {
  alert(event.data);
};
```

- ▶ Server: 通常のレスポンスとして **text/event-stream** を返却して SSE 接続を確立

```
< HTTP/1.1 200 OK
< cache-control: no-store
< content-type: text/event-
stream
< x-accel-buffering: no
< Transfer-Encoding: chunked
```



イベントストリーム処理



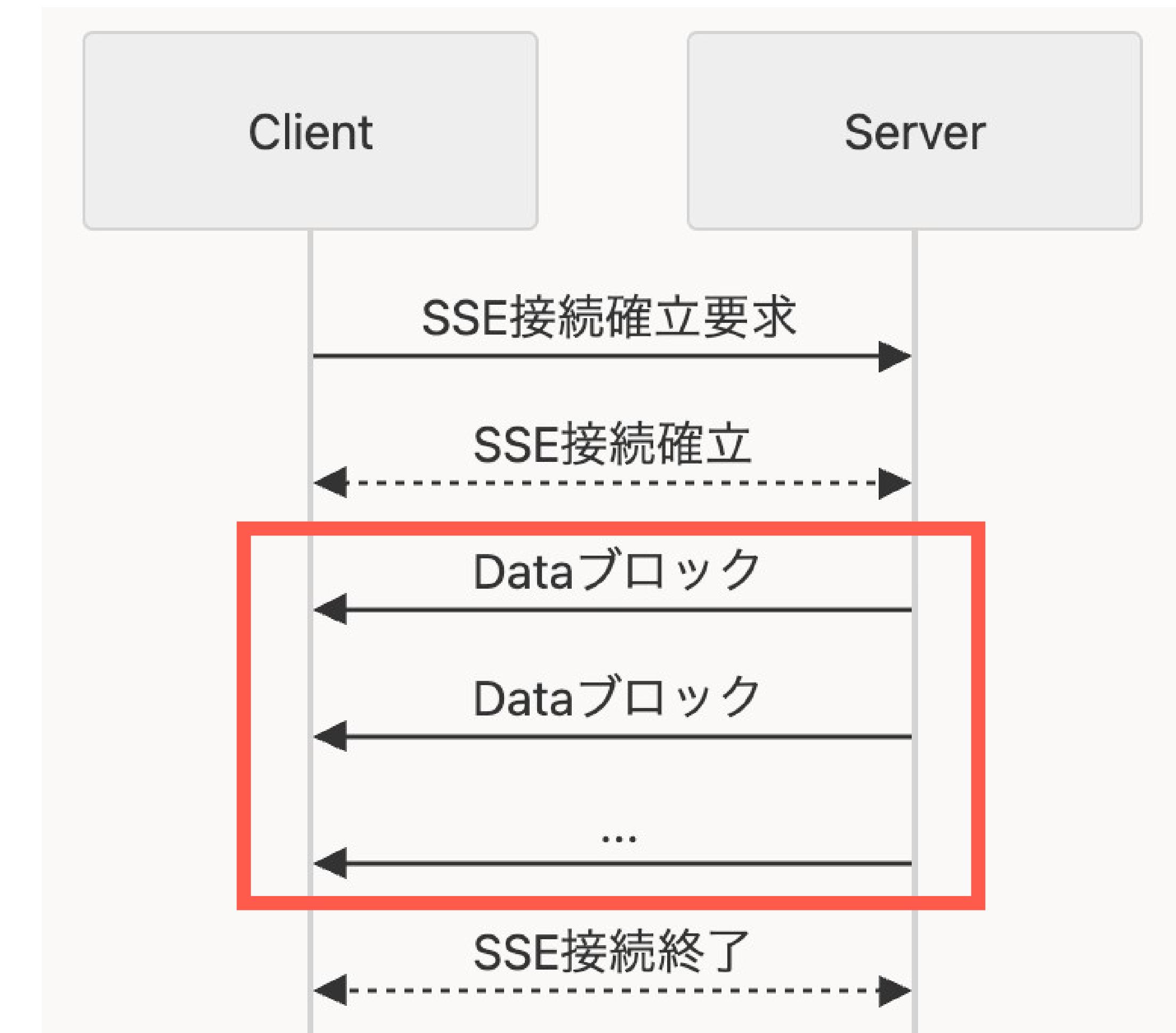
- ▶ ストリームの文字列は UTF-8
- ▶ 行指向のメッセージフォーマット
- ▶ 空行区切り `(＼n or＼r＼n)`

Example:

```
data: first event
id: 1

data: second event
id

data: third event
```



行の種別は以下の通り

行が空白（空白行）

- ▶ デリミタとして扱い、イベントを処理する

行がコロンから始まっている

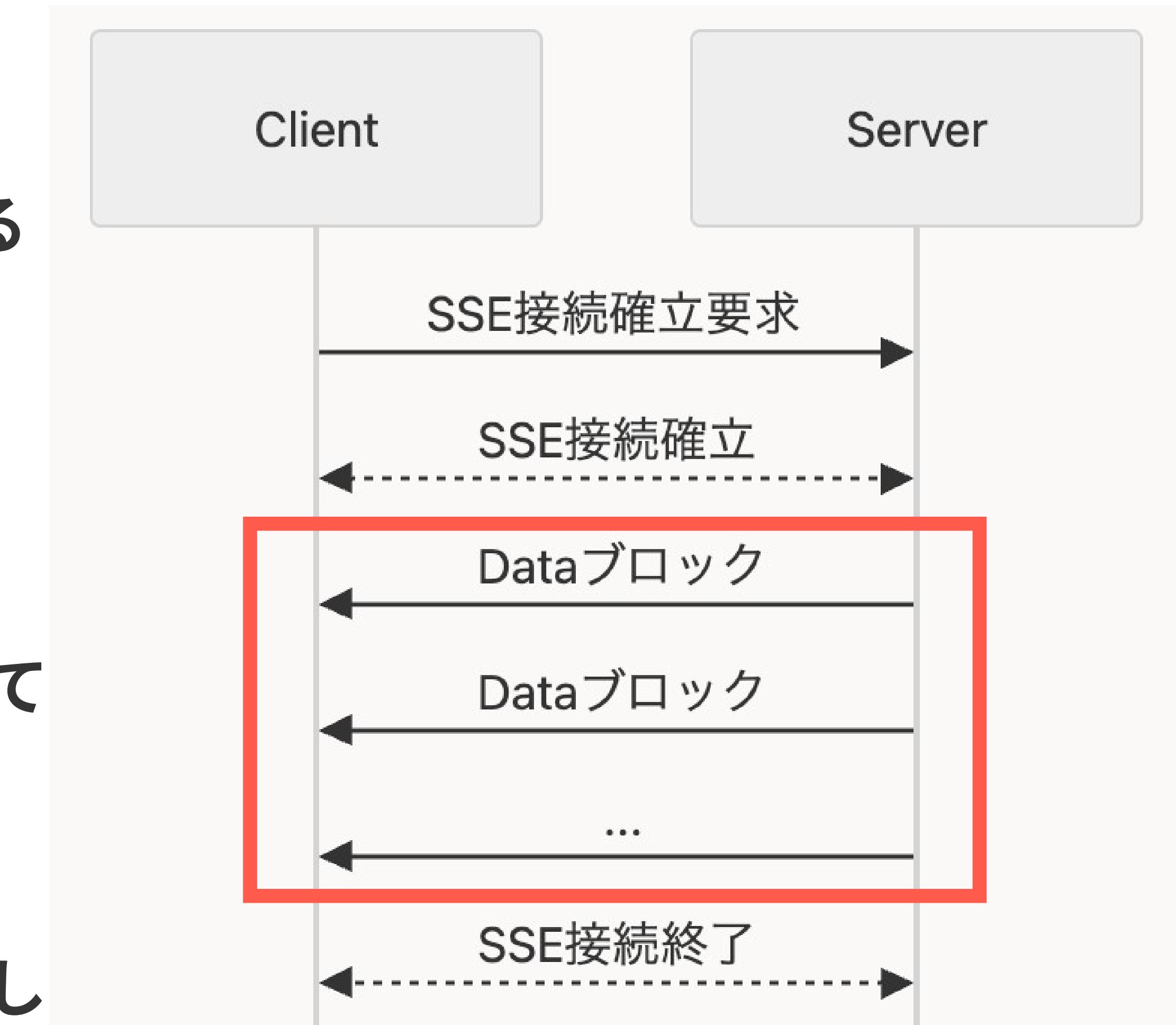
- ▶ 無視される（ハートビートに便利）

行にコロンが含まれている場合

- ▶ コロンによる Key (Field) と Value として扱う（後述）

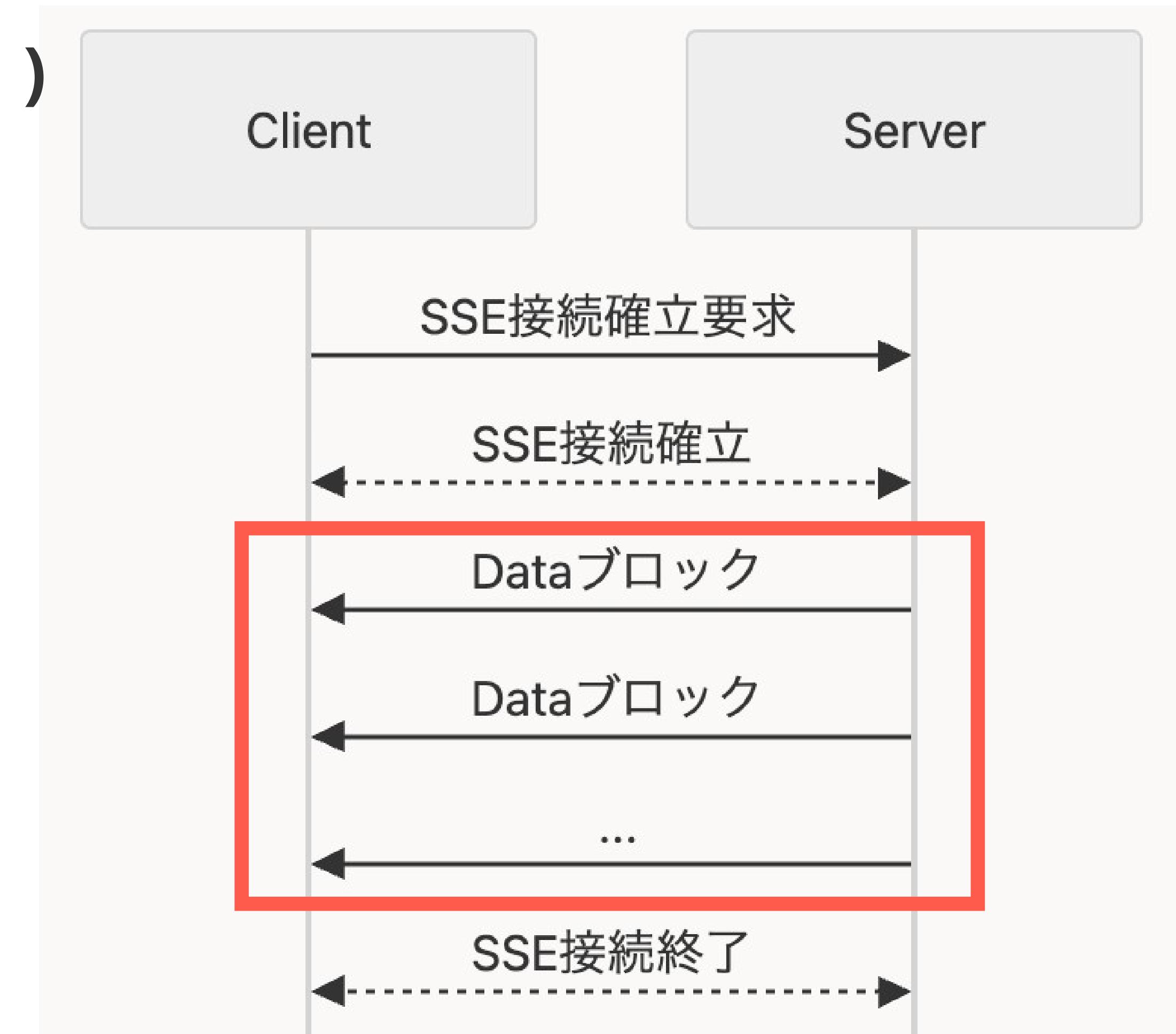
それ以外

- ▶ 行全体を Field として空文字を Value として扱う



- ▶ 許可される Field は以下（他は無視される）
 - ▶ **event**: 任意のイベントタイプ
 - ▶ **data**: 任意のデータ
 - ▶ **id**: 一意な ID
 - ▶ **Last-Event-ID** で再接続時の再開ポイントとして使われる
 - ▶ **retry**: 再接続時間（ミリ秒）

その他の細かなルールは仕様を参照されたい



例：

コロンからはじまるメッセージは無視される
このブロックでは何も発生しない

: test stream

data: first event

id: 1

data: second event

id

data: third event

Client

Server

SSE接続確立要求

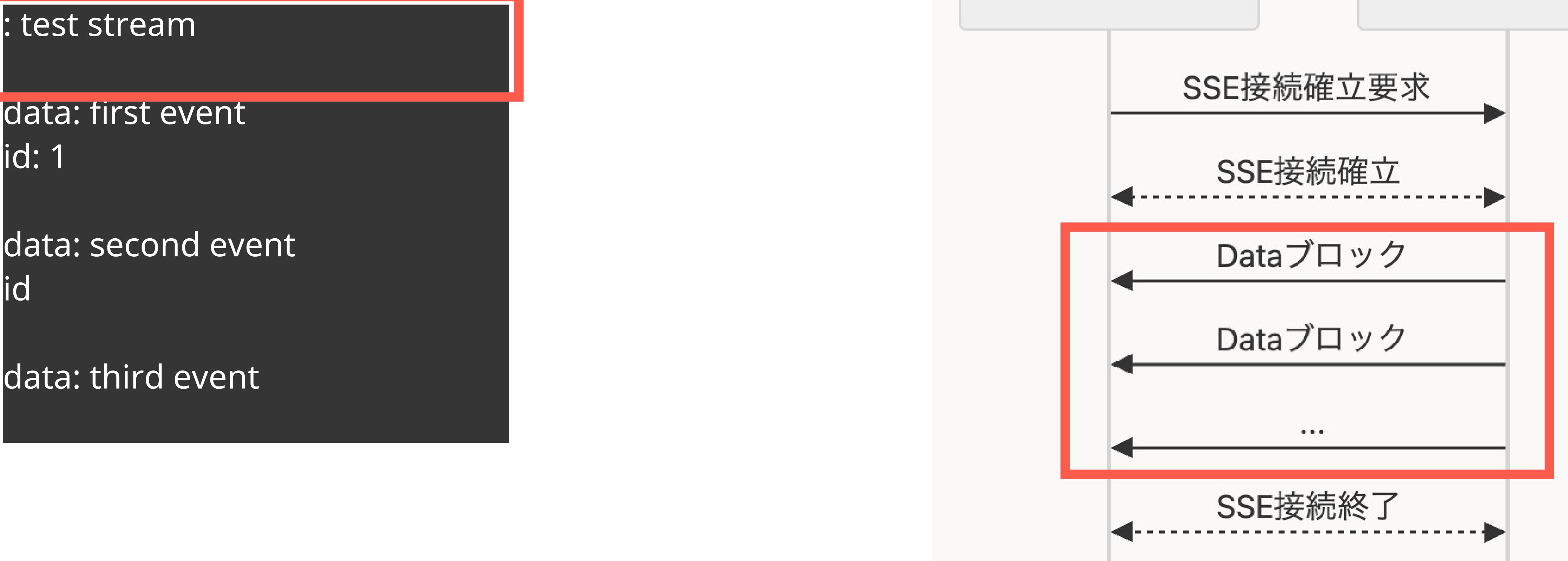
SSE接続確立

Data ブロック

Data ブロック

...

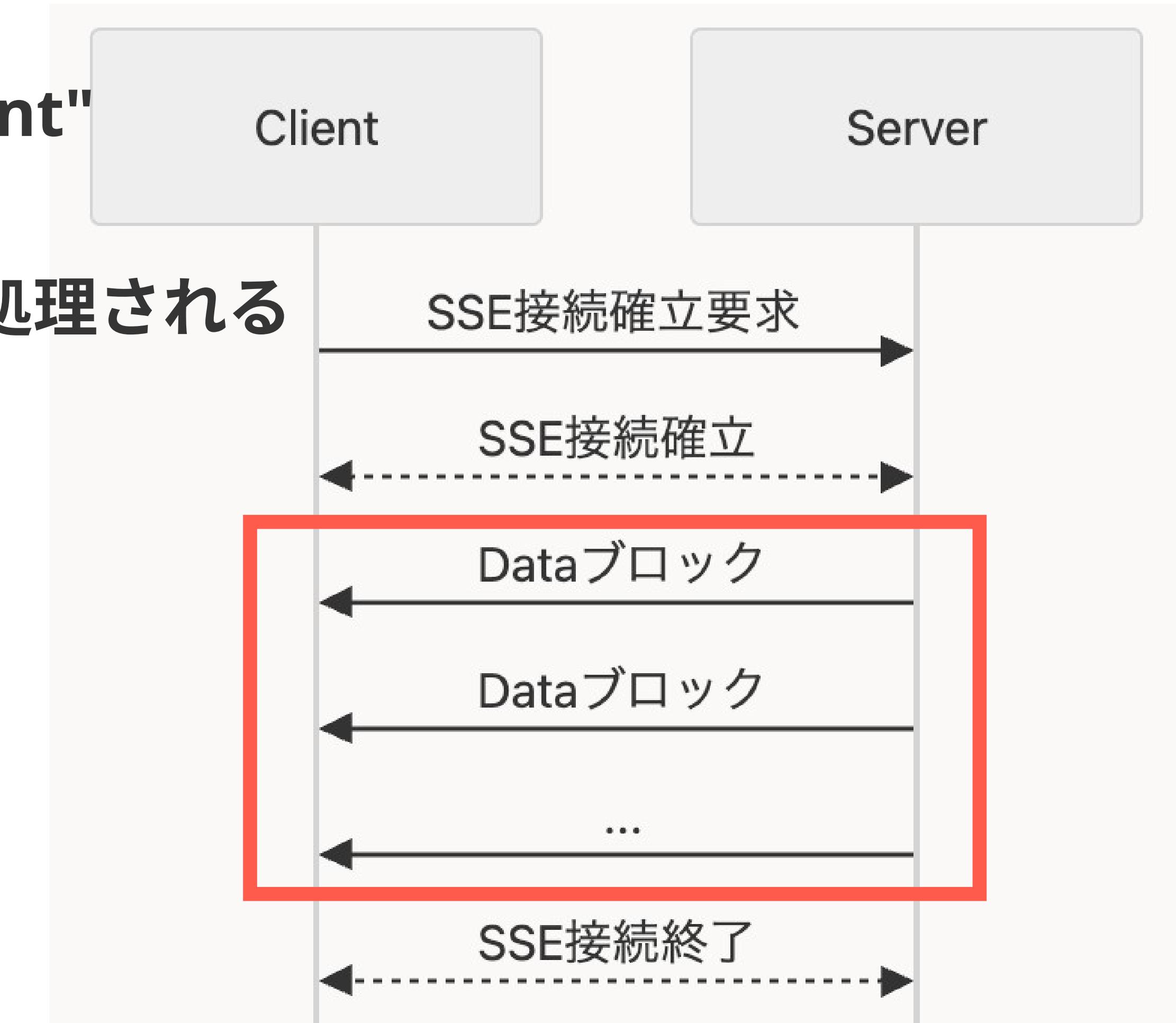
SSE接続終了



例：

```
: test stream
data: first event
id: 1
data: second event
id
data: third event
```

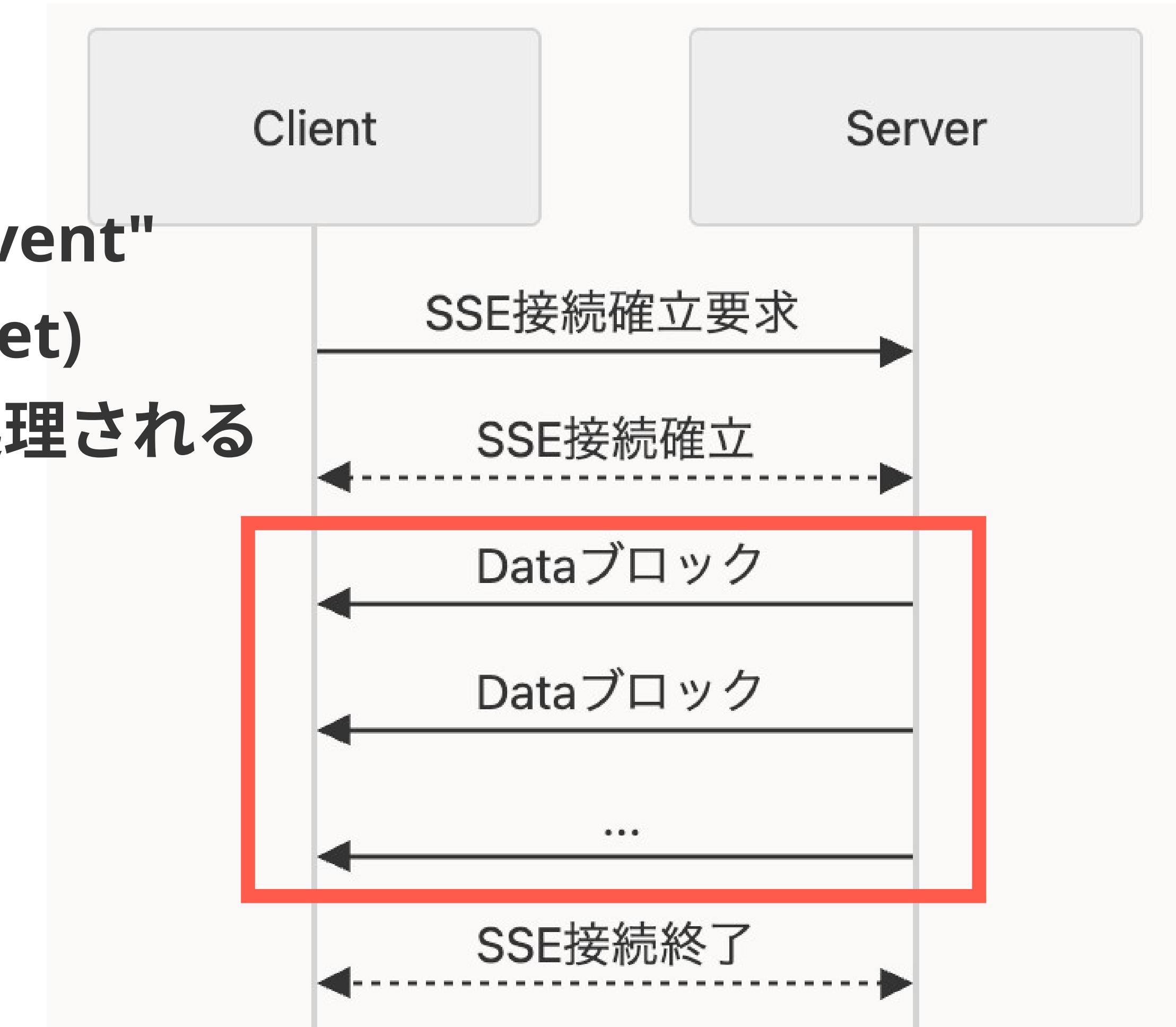
data が "first event"
id が "1"
としてブロックが処理される



例：

```
: test stream
data: first event
id: 1
data: second event
id
data: third event
```

data が "second event"
id が空文字列 (unset)
としてブロックが処理される



例：

```
: test stream
```

```
data: first event
```

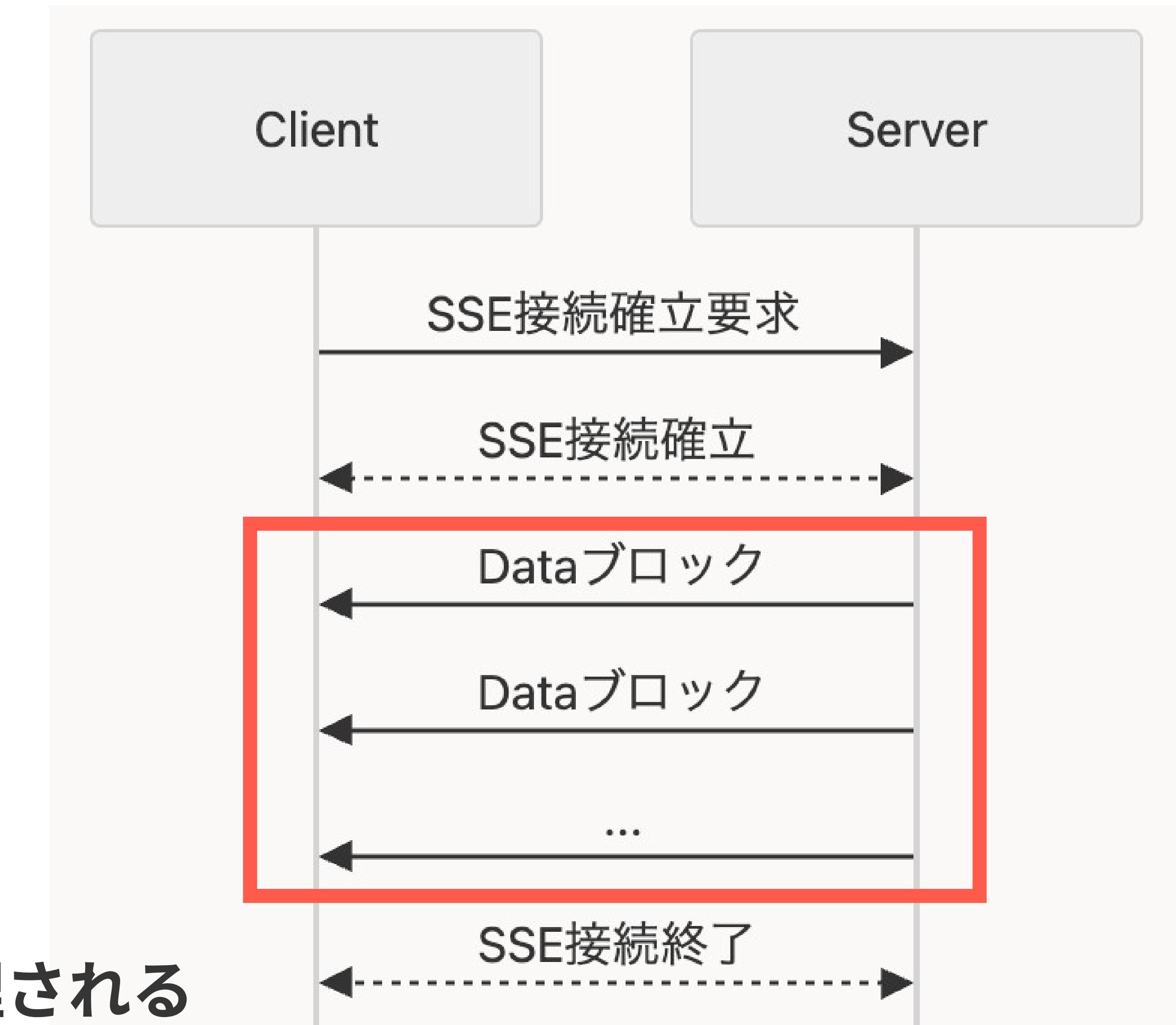
```
id: 1
```

```
data: second event
```

```
id
```

```
data: third event
```

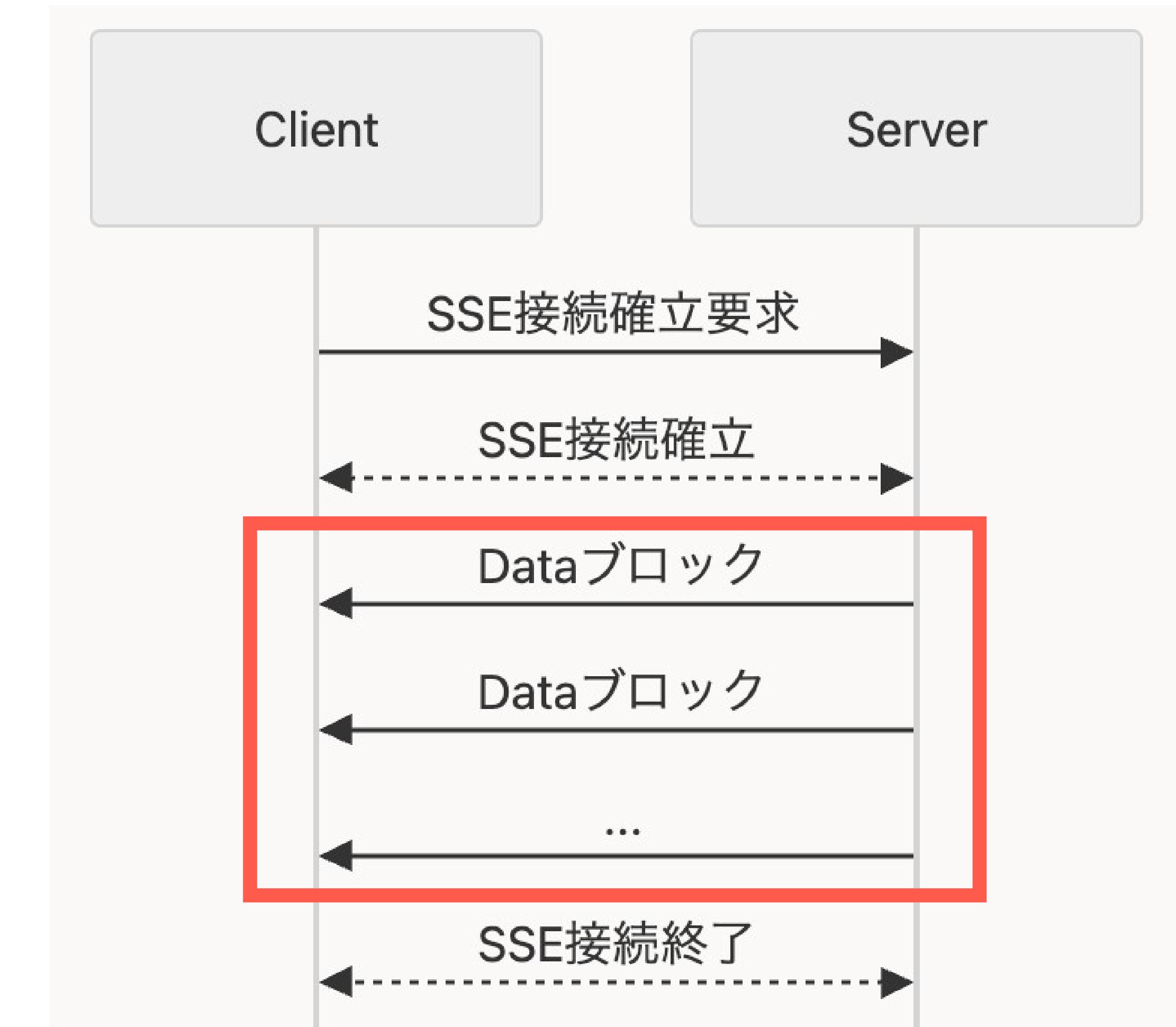
data が "third event"
としてイベントブロックが処理される
空行で終端されていないとイベントが発火しない



- ▶ クライアントは空行をデリミタとしたイベントブロックを 1 つの単位として処理する
- ▶ **data** が複数ある場合は改行 `\n` で連結されて渡される

```
data: uno
data: dos
data: tres
```

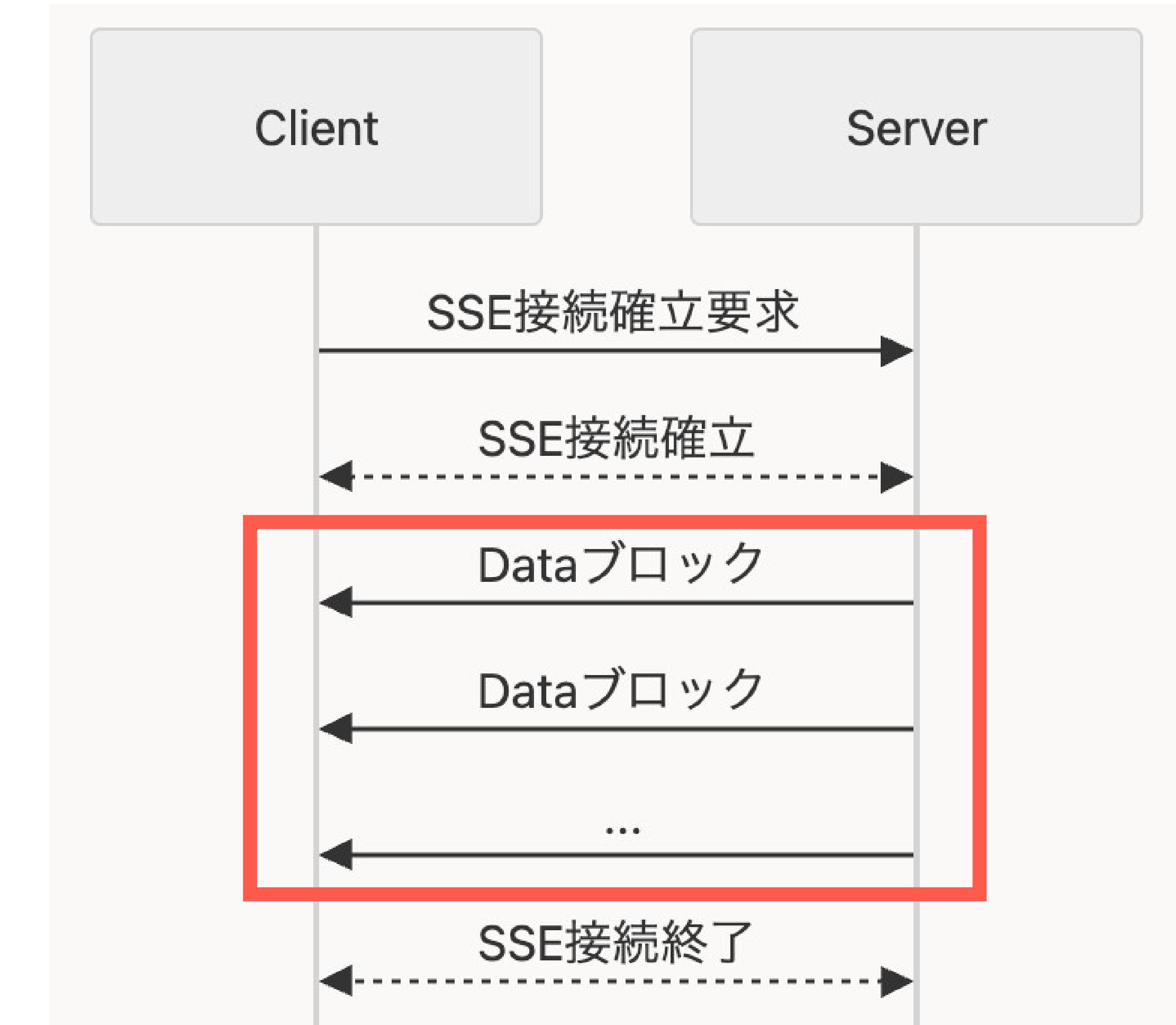
```
var source = new EventSource('updates.cgi');
source.onmessage = function (event) {
  alert(event.data);
};
```



- ▶ クライアントは空行をデリミタとしたイベントブロックを 1 つの単位として処理する
- ▶ `data` が複数ある場合は改行 `\n` で連結されて渡される

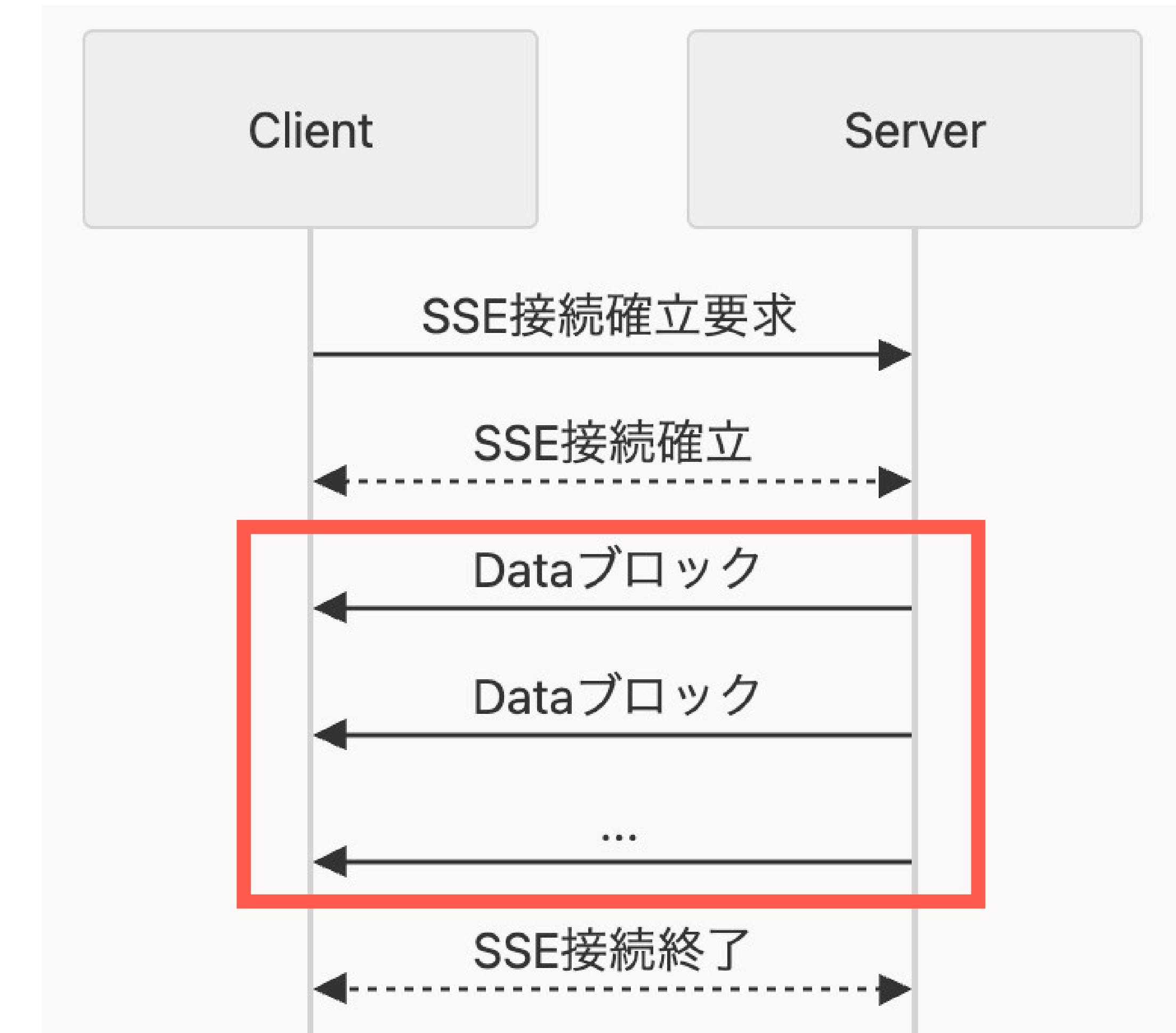
```
data: uno
data: dos
data: tres
```

```
var source = new EventSource('updates.cgi');
source.onmessage = function (event) {
  alert(event.data);
};
```

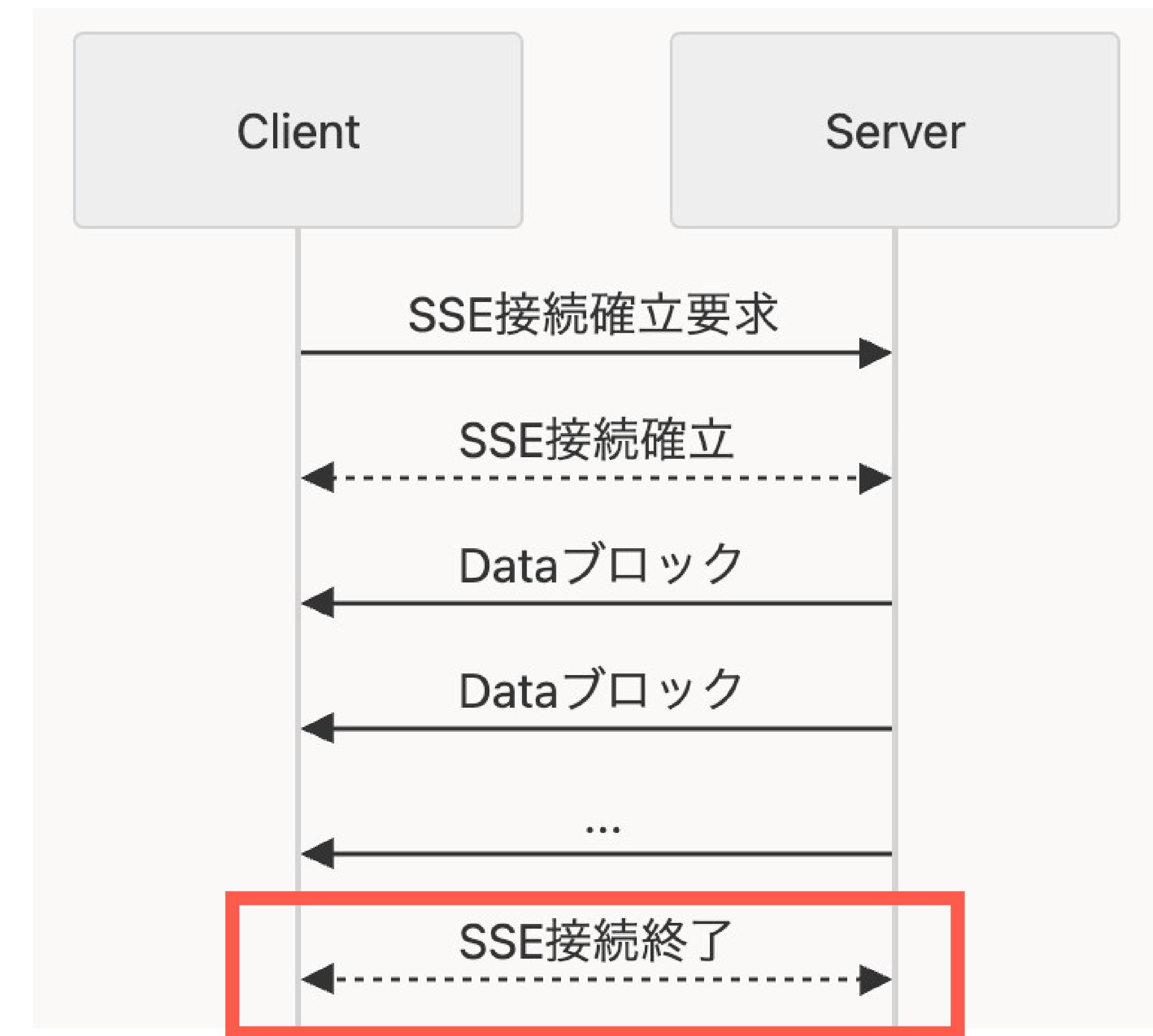


ここにはブロック内のデータが `'uno\ndos\ntres'` として来る

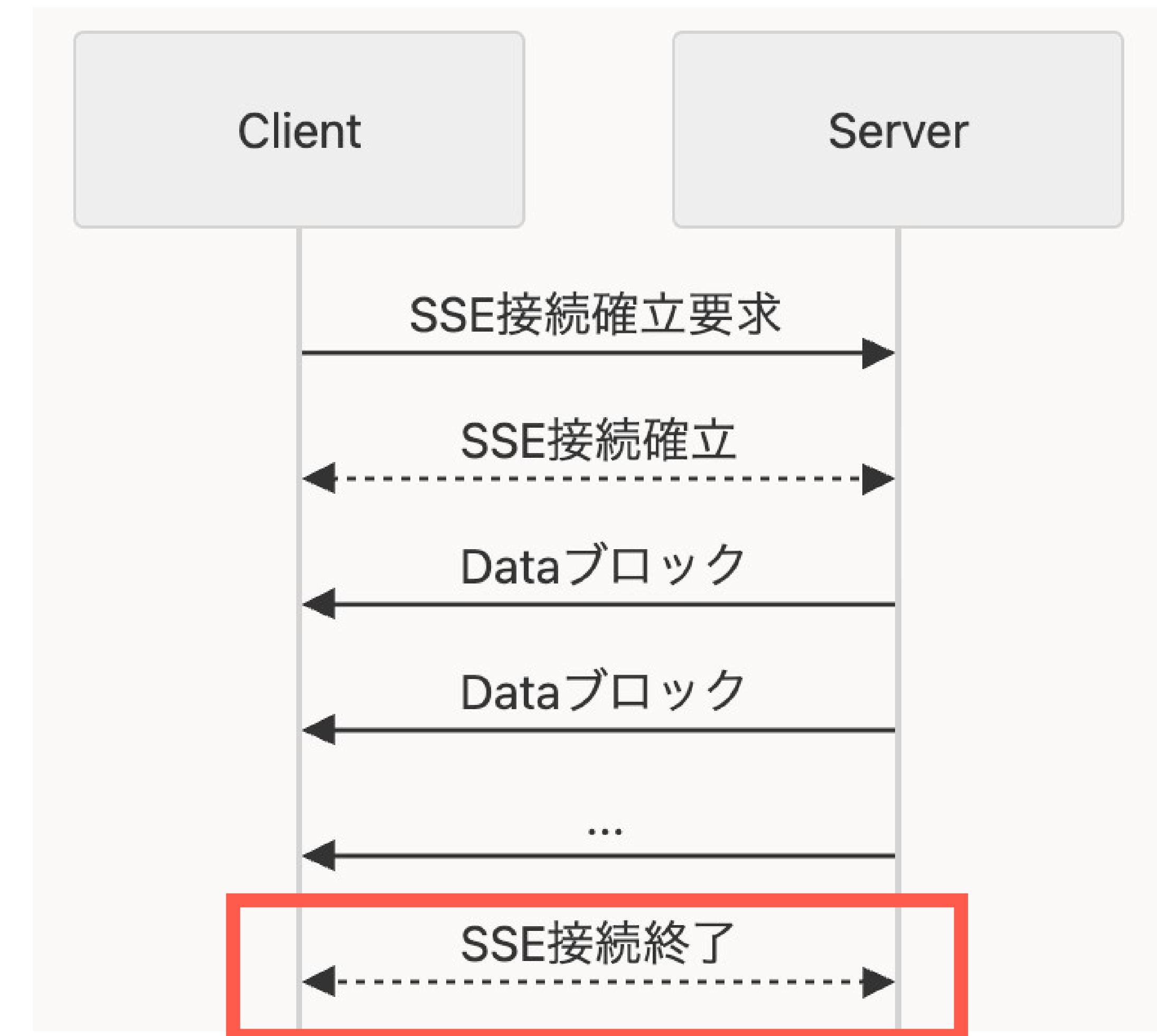
- ▶ クライアントは逐次来る **event, data, id** を処理して所望の機能を実現する
- ▶ 実用例 : **data** に JSON が入ってくるので クライアントはそれを都度レンダリングする
- ▶ **retry** は基本的にクライアントライブラリがケアする



接続終了処理



- ▶ クライアントから明示的に SSE 接続を **close** する
 - ▶ 例 : **eventSource.close()**
- ▶ サーバーから SSE 接続を **close** する
 - ▶ 例 : **sse.close**
- ▶ 適確に **close** されなかった場合は クライアントから再接続が行なわれる



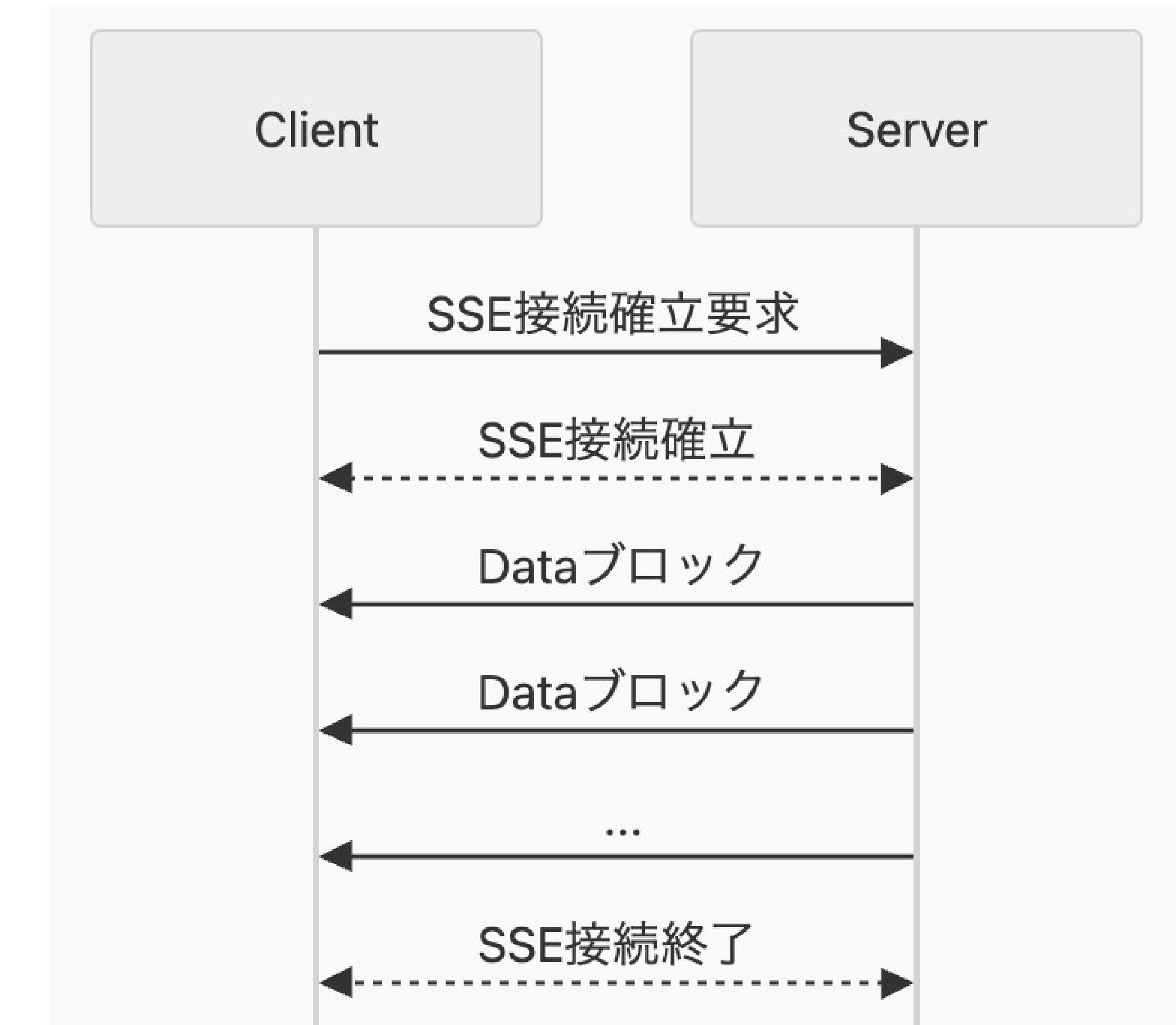
基本的にクライアントライブラリはこのへんをよしなにやってくれる

- ▶ ブラウザにはもう実装が存在している
- ▶ **EventSource** インターフェイス

	Chrome	Edge	Firefox	Opera	Safari	Chrome Android	Firefox for Android	Opera Android	Safari on iOS	Samsung Internet	WebView Android	WebView on iOS	Deno
EventSource	✓ 6	✓ 79	✓ 6	✓ 11	✓ 5	✓ 18	✓ 45	✓ 11	✓ 5	✓ 1	✓ 4.4	✓ 5	✓ 1.38

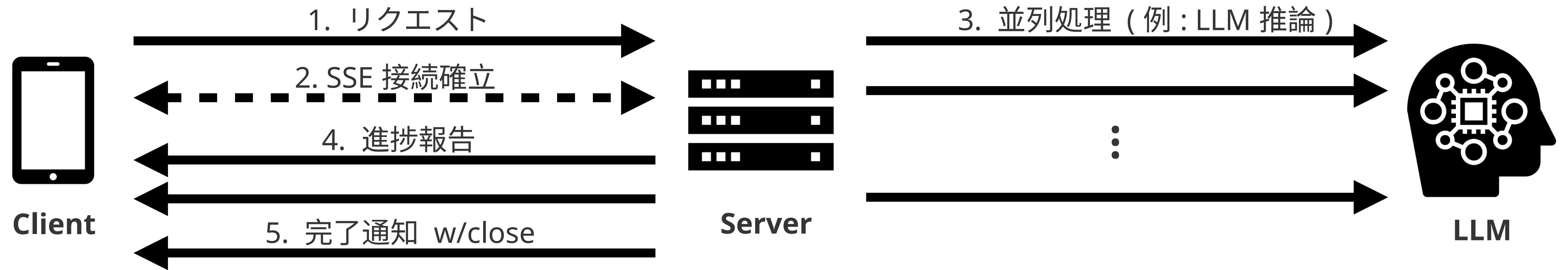
出展 : <https://developer.mozilla.org/en-US/docs/Web/API/EventSource>

- ▶ Native App 用のクライアントも複数ある

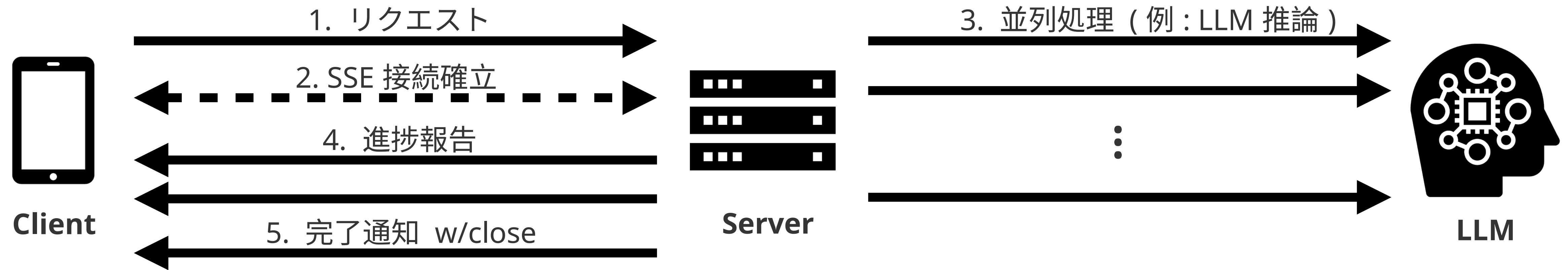


SSE と非同期処理は相性が良さそうに思いませんか

サンプルシナリオ：
サーバー側で非同期で重い処理を並列化し、その進捗をクライアントに知らせる

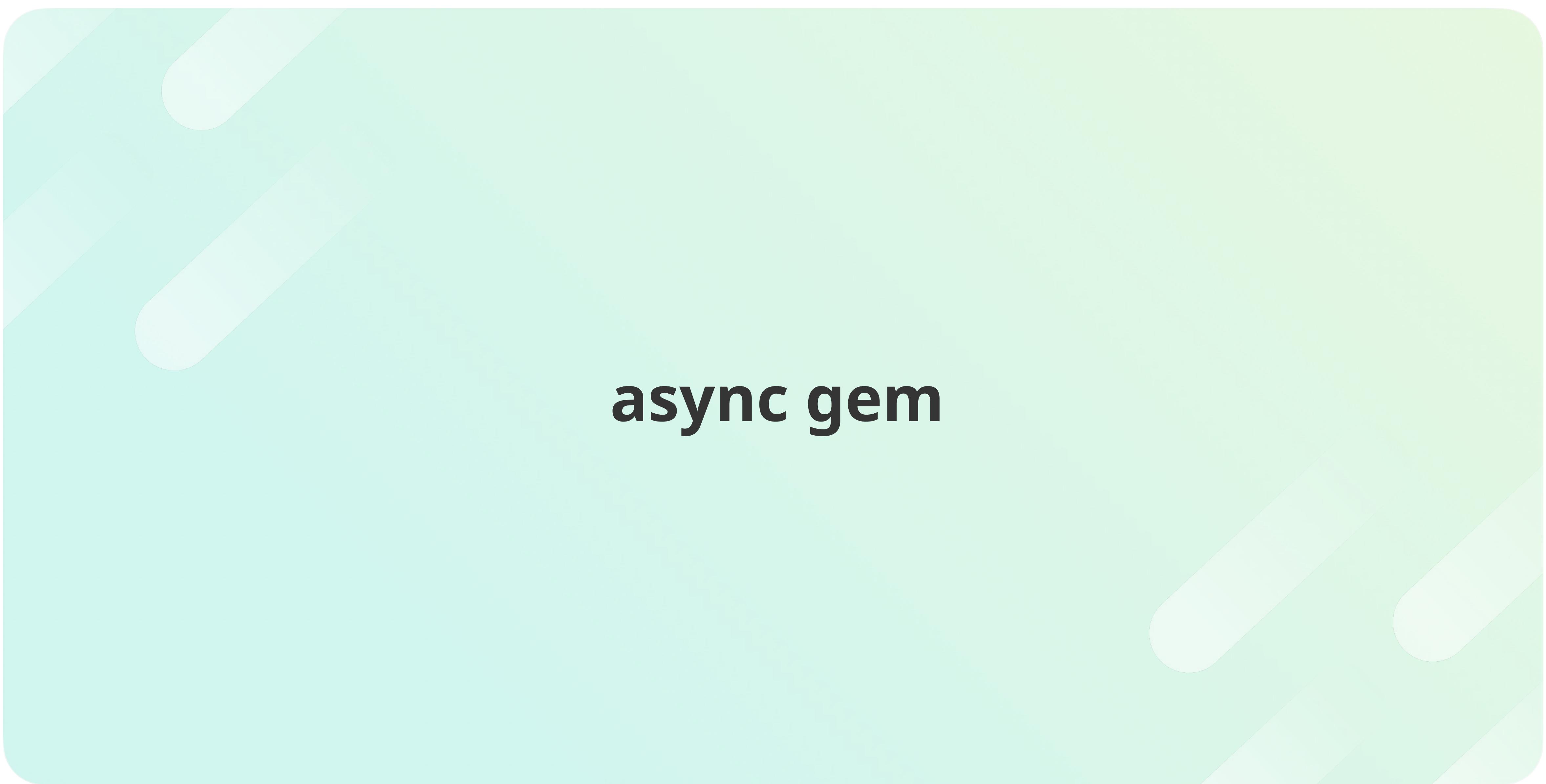


サンプルシナリオ：
サーバー側で非同期で重い処理を並列化し、その進捗をクライアントに知らせる



👉 I/O 待ちのような時間のかかる処理には一般に非同期処理が有効

SSE による表面的な体験の改善に加え
非同期処理による本質的なパフォーマンスの改善が見込める

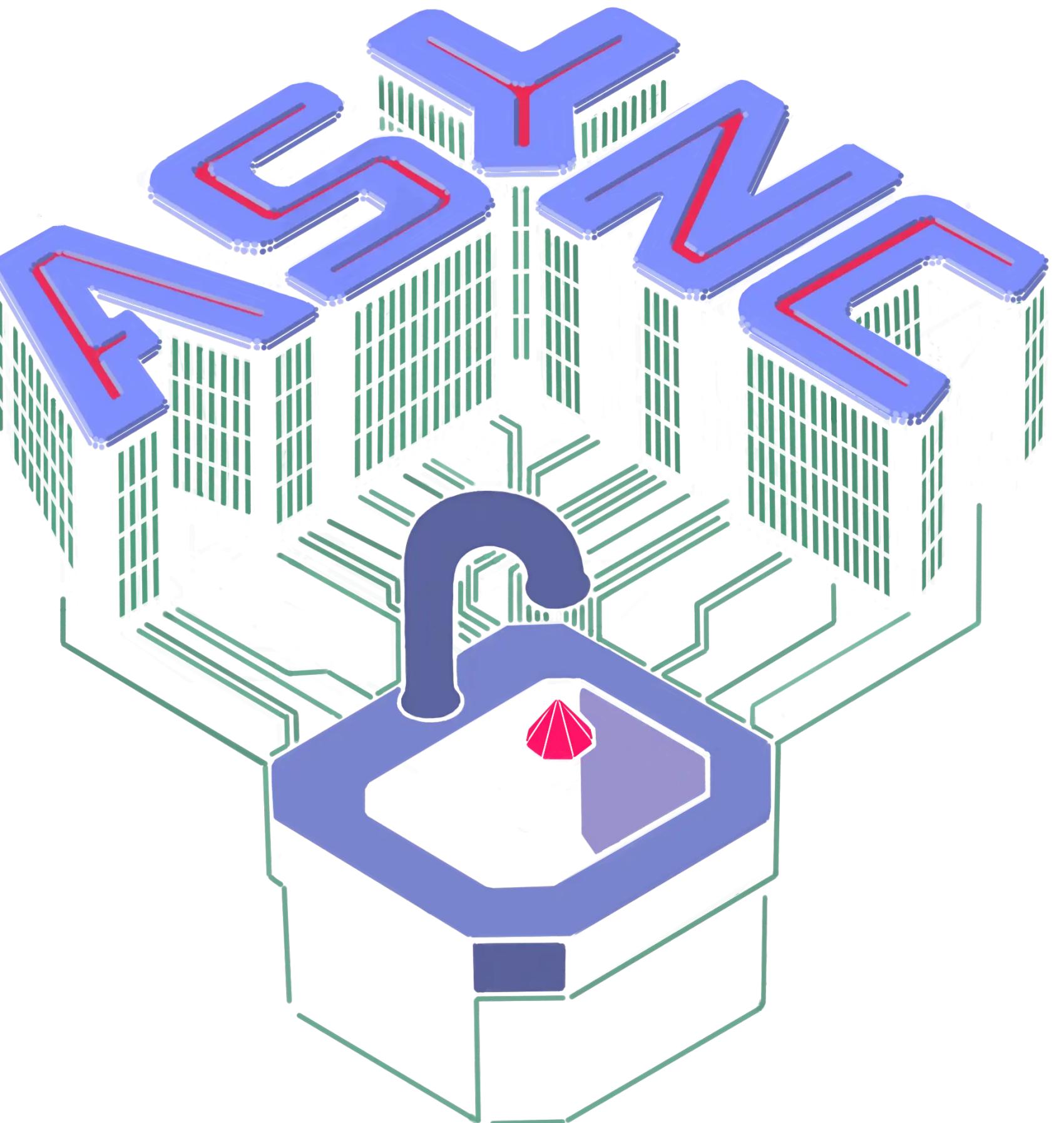


async gem

socketry/async

”
Async is a composable asynchronous I/O framework for Ruby based on [io-event](#).

Fiber をベースとしたイベントドリブン I/O を提供
(個人的には純粹に書き味が良く、好み)



Thread

- ▶ GVL の影響を受けるが、I/O 待ちなどは並列化できる
- ▶ メモリリソースを食う：大量に `spawn` できない

Fiber

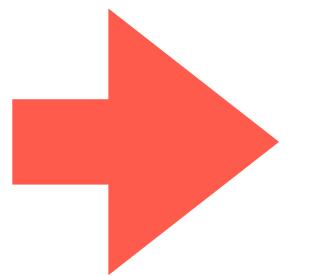
- ▶ 協調的（イベントループとノンブロッキング I/O）
- ▶ メモリリソースに対する優位性がある
- ▶ CPU バウンドな処理は並列化できない
- ▶ ブロッキング処理が入ると全体をブロックする
- ▶ 関連するコンポーネントが非同期 I/O ・ Fiber aware である必要がある

Thread

- ▶ GVL の影響を受けるが、I/O 待ちなどは並列化できる
- ▶ メモリリソースを食う：大量に `spawn` できない

Fiber

- ▶ 協調的（イベントループとノンブロッキング I/O）
- ▶ メモリリソースに対する優位性がある
- ▶ CPU バウンドな処理は並列化できない
- ▶ ブロッキング処理が入ると全体をブロックする
- ▶ 関連するコンポーネントが非同期 I/O ・ Fiber aware である必要がある



I/O 待ち過多な
タイプのアプリ
では Thread より
スケールする
(可能性が高い)

SSE と組み合わせた実装パターン

Async::Queue との相性が良さそう

- ▶ `#start` でタスクを並列処理
- ▶ 完了したら SSE のメッセージを Async::Queue に enqueue
- ▶ `#next` で SSE メッセージを queue から取得

```
require 'async'
require 'async/queue'

END_OF_STREAM = Object.new # <= sentinel

class SomethingAsyncRunner
  def initialize
    @queue = Async::Queue.new
  end

  def start
    Async do |task|
      [:one, :two, :three].map do |v|
        task.async do
          sleep(rand(3))
          @queue.enqueue({event: :in_progress, body: {event: :in_progress, value: v}})
        end
      end.each(&:wait)
      @queue.enqueue({event: :finished, body: {event: :finished}})
    end
  end

  def next
    event = @queue.dequeue
    raise StopIteration if event.equal?(END_OF_STREAM)
    event
  end

  private

  def finalize
    @queue.enqueue(END_OF_STREAM)
  end
end
```

SSE と組み合わせた実装パターン

こんな感じでイテレータ的に扱うというパターン

```
class MyController < ActionController::Base
  include ActionController::Live

  def index
    response.headers['Content-Type'] = 'text/event-stream'
    sse = SSE.new(response.stream, retry: 250)

    # NOTE: `id` を使う場合はロジック側で上手くハンドリングする必要がありそう。
    runner = SomethingAsyncRunner.new

    begin
      Async do
        runner.start

        loop do
          resp = runner.next
          sse.write(resp[:body], event: resp[:event])
        end
      end
    ensure
      sse.close
    end
  end
```

```
require 'async'
require 'async/queue'

END_OF_STREAM = Object.new # <= sentinel

class SomethingAsyncRunner
  def initialize
    @queue = Async::Queue.new
  end

  def start
    Async do |task|
      [:one, :two, :three].map do |v|
        task.async do
          sleep(rand(3))
          @queue.enqueue({event: :in_progress, body: {event: :in_progress, value: v}})
        end
      end.each(&:wait)
      @queue.enqueue({event: :finished, body: {event: :finished}})
    end
  end

  def next
    event = @queue.dequeue
    raise StopIteration if event.equal?(END_OF_STREAM)
    event
  end

  private

  def finalize
    @queue.enqueue(END_OF_STREAM)
  end
end
```

Barrier と Semaphore を組み合わせると良い

- ▶ Semaphore に最大並列実行数と Barrier を与える
- ▶ `Semaphore#async` の中で処理すると実行数を制限できる
- ▶ `Barrier#wait` で完了待ち

```
Async do
  barrier = Async::Barrier.new
  semaphore = Async::Semaphore.new(num_of_parallel_tasks, parent: barrier)

  items.each do |item|
    semaphore.async(parent: barrier) do
      # do something
      # ...
    end
  end

  barrier.wait
end
```

well-documented で最高

- ▶ <https://socketry.github.io/async/guides/tasks/index.html#combining-a-barrier-with-a-semaphore>
- ▶ <https://socketry.github.io/async/guides/best-practices/index.html#use-a-top-level-sync-to-denote-the-root-of-your-program>

Task#with_timeout を使うと良い

- ▶ タイムアウトしうる処理をコードブロックで与える
- ▶ タイムアウトしたら Async::TimeoutError が上がる

```
Async do
  Async::Task.current.with_timeout(TIMEOUT_SECS) do
    # do something
  rescue Async::TimeoutError
    # do something
  end
end
```

well-documented で最高

- ▶ <https://socketry.github.io/async/guides/tasks/index.html#timeouts>
- ▶ <https://socketry.github.io/async/guides/best-practices/index.html#use-a-top-level-sync-to-denote-the-root-of-your-program>

SSE と組み合わせた実装パターン（イレギュラー）

async task 内から Active Record (mysql2) のコネクションを **borrow** すると、動作中の Thread 内のコネクションを借用することになる。

借りてきた（单一 Thread に割り当てられた）コネクションを Fiber 橫断で使い回すとリソース競合が発生してクラッシュしてしまう。

```
thread_pool = Concurrent::FixedThreadPool.new(num_of_parallel_tasks)
active_record_safe_zone = ->(&block) do
  # ActiveRecord::Base.connection_pool.with_connectionを強制することでスレッドによるコネクションのリークを防ぐ。
  future = Concurrent::Future.execute(executor: thread_pool) { ActiveRecord::Base.connection_pool.with_connection(&block) }
  future.value # ここで同期を取る
end
```

上記のようにスレッドプールとコネクションプールを併用し、**spawn** された 1 Thread が 1DB コネクションを適切に取得する **wrapper** を使うことで、**async task** 内から DB を並行に触れるような解決を試みた。

他に何か良いアイデアがあれば知りたい…… (Thread を作ると Fiber の良さが減るので)

**async には SSE と組み合わせて使うための
機能が全部揃っていてすばらしい**

すべてが解決
めでたしめでたし

と思いきや実動までは様々なブロックがある
(あった)

- ▶ 標準仕様では `Cache-Control: no-store` を付けよ、というふうに書いてある
- ▶ MDN のドキュメントでは `Cache-Control: no-cachè` を付けている
 - ▶ どっちやねん (no-cache で良さそうだが……)
- ▶ React のプロキシを通すと SSE (Server-sent events) のレスポンスを受け取れない
#React - Qiita を参考にすると `no-transform` を付けたほうが良さそうに見える
 - ▶ なので `Cache-Control: no-cache, no-transform` を付与することで動かしている

- ▶ リバースプロキシのバッファリングを抑制するために **X-Accel-Buffering: no** も設定しておく
- ▶ このあたりの調整はリバースプロキシの設定と同期を取る必要あり
- ▶ 実例：

```
< HTTP/1.1 200 OK
< cache-control: no-cache, no-transform
< content-type: text/event-stream
< x-accel-buffering: no
< Transfer-Encoding: chunked
```

例：nginx.conf

- ▶ **location** に右のような設定を付与する
 - ▶ 前掲のレスポンスヘッダと重複する箇所はあるが……
 - ▶ プロトコル的な特性から SSE については HTTP のバージョンを 1.1 に固定しておくほうが安全
 - ▶ リバースプロキシが多段であった場合は経路すべての設定をする必要がある
 - ▶ 一部のリバースプロキシで設定が漏れていて動かなかつた経験あり (1 敗)

```
# SSEを利用するため以下のように設定が必要となっている {{{  
proxy_set_header Connection '';  
proxy_http_version 1.1;  
chunked_transfer_encoding off;  
proxy_buffering off;  
proxy_cache off;  
# }}}}
```

ブロック色々：便利な `ActionController::Live` だが……

○`include ActionController::Live` したコントローラーで `application/json` のような `content-type` が「普通な」レスポンスを返却しようとすると刺さる

- ▶ `Action` 内で「SSE のような streaming」を返却する時はコード内で明示的に`close` してリソースを返却するチャンスがある
- ▶ 一方、普通のリソースを普通に返却する (e.g. `render json ...`) と`close` が呼ばれる機会が無いためリソースが取られっぱなしに

👉 `puma thread` が `busy` になってアプリケーション全体がストールする

⇒ 従って SSE とそれ以外とで Controller Class を分ける必要がある
(Rails 的にちょっと微妙な見た目に……?)

committee gem の Response Body Validation Middleware を使うと刺さる

- ▶ 正しく言うとレスポンスボディを全部返し終えるまでミドルウェアでバッファしてしまうのでストリーミング処理にならない
- ▶ **puma thread** が **busy** になってアプリケーション全体がストールする（再登場）
- ▶ 詳細は過去の資料 : [Committee::Middleware::ResponseValidation](#) で Streaming Response Body を処理すると刺さります

```
def response_validate(status, headers, response, test_method = false)
  full_body = ""
  response.each do |chunk|
    full_body << chunk
  end

  parse_to_json = if validator_option.parse_response_by_content_type
    content_type_key = headers.keys.detect { |k| kcasecmp?('Content-Type') }
    headers.fetch(content_type_key, nil)&.start_with?('application/json')
  else
    true
  end

  data = if parse_to_json
    full_body.empty? ? {} : JSON.parse(full_body)
  else
    full_body
  end

# TODO: refactoring name
strict = test_method
Committee::SchemaValidator::OpenAPI3::ResponseValidator.
  new(@operation_object, validator_option).
  call(status, headers, data, strict)
end
```

アツ

- ▶ committee に限らずレスポンスボディ全体をバッファするようなミドルウェアでは同様の問題が発生する可能性がある
- ▶ サーバー側だけではなくクライアント側でも起こる可能性があるので注意
- ▶ 例 : Android の HttpLoggingInterceptor

OpenAPI と SSE の相性があまり良くない

- ▶ OpenAPI が RESTful な API 定義を記述することを主眼としているため Streaming な SSE との相性があまり良くない
- ▶ クライアントの自動生成は今ある一般的なツールでは困難

SSE でハマった時のトラブルシューティング

リクエストヘッダとレスポンスヘッダを真っ先に見る

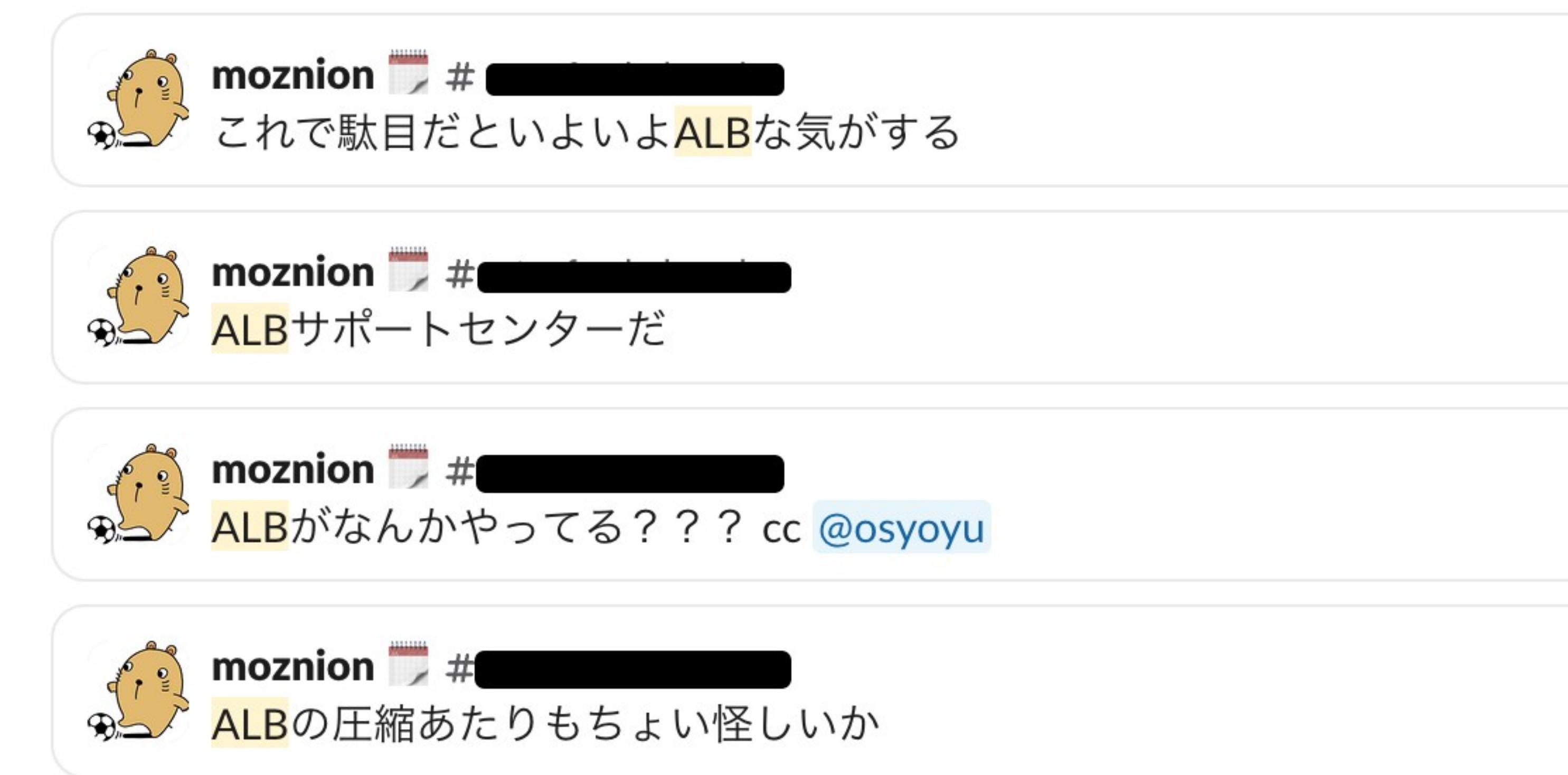
- ▶ ヘッダが違っていると基本的には正常に動かない
- ▶ (たまに実装依存でうっかり動いたりしてしまうことがあるが……)

- ▶ リクエストとレスポンスが通る経路すべてをチェックする
- ▶ 例えばリバースプロキシがあるのにアプリサーバーとだけ疎通試験しても意味が無い

- ▶ リクエストとレスポンスが通る経路すべてをチェックする
- ▶ 例えばリバースプロキシがあるのにアプリサーバーとだけ疎通試験しても意味が無い
- ▶ (経験上) AWS の ALB は原因にならない



- ▶ リクエストとレスポンスが通る経路すべてをチェックする
- ▶ 例えばリバースプロキシがあるのにアプリサーバーとだけ疎通試験しても意味が無い
- ▶ (経験上) AWS の ALB は原因にならない



- ▶ チャンクサイズで切られて SSE のコンテンツは送られてくることに留意する
 - ▶ SSE のイベントブロック単位で送られてくるわけではなく、puma や nginx 等で設定されるチャンクサイズに従って送られてくる
 - ▶ クライアントはレスポンスボディをストリーミングで受け取り、それをバッファしていきながらデリミタが来た時に SSE の 1 イベントブロックとして parse する必要がある

- ▶ **tcpdump (パケットキャプチャ) を見られるのが一番良い**
- ▶ いろいろと面倒であるのはそう
 - ▶ **Amazon ECS のようなコンテナ環境に差し込む方法**
 - ▶ **TLS に阻まれる**



...

tcpdumpをシュッと取りたい時、本当にその時だけTLSをちょっとだけ恨むことがある.....

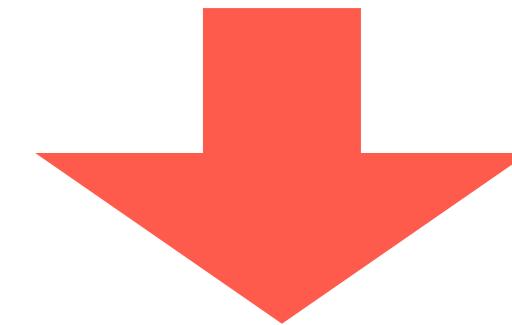
午前11:25 · 2025年5月27日 · 1,021 件の表示

ブロッカーを取り除いていく取り組み

ブロッカー色々：便利な `ActionController::Live` だが……

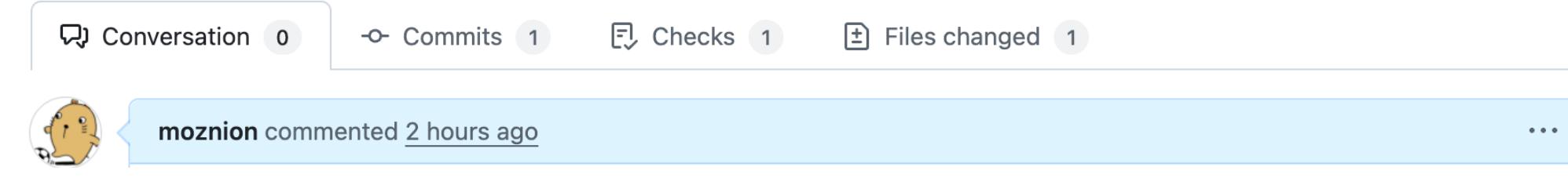
- `include ActionController::Live` したコントローラーで `application/json` のような `content-type` が「普通な」レスポンスを返却しようと刺さる
 - ▶ `puma thread` が `busy` になってアプリケーション全体がストールする

- `include ActionController::Live` したコントローラーで `application/json` のような content-type が「普通な」レスポンスを返却しようと刺さる
 - ▶ puma thread が busy になってアプリケーション全体がストールする



Add documentation on how to prevent thread exhaustion in
ActionController::Live #55763

[Open](#) moznion wants to merge 1 commit into `rails:main` from `moznion:doc_action_thread_exhaustion`



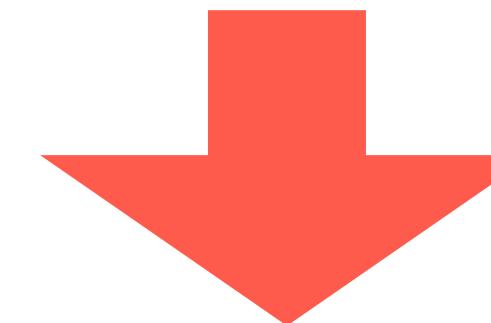
<https://github.com/rails/rails/pull/55763>

committee gem の Response Body Validation Middleware を使うと刺さる

- ▶ 正しく言うとレスポンスボディを全部返し終えるまでミドルウェアでバッファしてしまうのでストリーミング処理にならない
- ▶ **puma thread** が **busy** になってアプリケーション全体がストールする（再登場）

committee gem の Response Body Validation Middleware を使うと刺さる

- ▶ 正しく言うとレスポンスボディを全部返し終えるまでミドルウェアでバッファしてしまうのでストリーミング処理にならない
- ▶ **puma thread が busy になってアプリケーション全体がストールする（再登場）**



Supports streaming response validation via `streaming_content_parsers` option #447

 Merged geemus merged 2 commits into `interagent:master` from `moznion:streamingg_support` on Aug 3

 Conversation 12  Commits 2  Checks 8  Files changed 7



moznion commented on Jun 30 · edited

Contributor

Reviewers

<https://github.com/interagent/committee/pull/447>

Usage Example:

```
use Committee::Middleware::ResponseValidation,  
  schema_path: 'docs/schema.json',  
  streaming_content_parsers: {  
    'text/event-stream' => ->(body) { body }, # Pass through SSE as string  
    'application/x-json-stream' => ->(body) { JSON.parse(body) } # Parse custom JSON stream  
  }
```

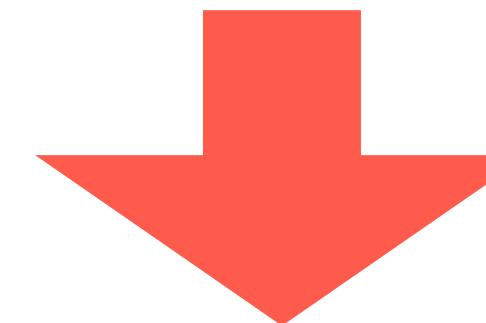
内部実装 (簡略):

```
if streaming_response?(headers)  
  response = Rack::BodyProxy.new(response) do  
    begin  
      validate(request, status, headers, response)  
      rescue %=> e  
        handle_exception(e, request.env)  
  
        raise e if @raise  
      end  
    end  
  else
```

内部的に Rack::BodyProxy を使用
詳しくは Pull Request を参照のこと

- ▶ OpenAPI と SSE の相性があまり良くない
 - ▶ OpenAPI が RESTful な API 定義を記述することを主眼としているため Streaming な SSE との相性があまり良くない
 - ▶ クライアントの自動生成は今ある一般的なツールでは困難

- ▶ OpenAPI と SSE の相性があまり良くない
 - ▶ OpenAPI が RESTful な API 定義を記述することを主眼としているため Streaming な SSE との相性があまり良くない
 - ▶ クライアントの自動生成は今ある一般的なツールでは困難



An SSE Proposal #4171

 Closed

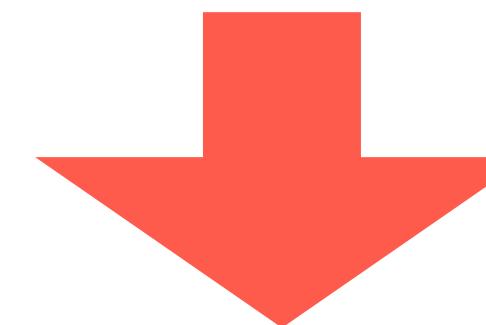
robertlagrant started this conversation in Enhancements



robertlagrant on Nov 1, 2024

<https://github.com/OAI/OpenAPI-Specification/discussions/4171>

- ▶ OpenAPI と SSE の相性があまり良くない
 - ▶ OpenAPI が RESTful な API 定義を記述することを主眼としているため Streaming な SSE との相性があまり良くない
 - ▶ クライアントの自動生成は今ある一般的なツールでは困難



An SSE Proposal #4171

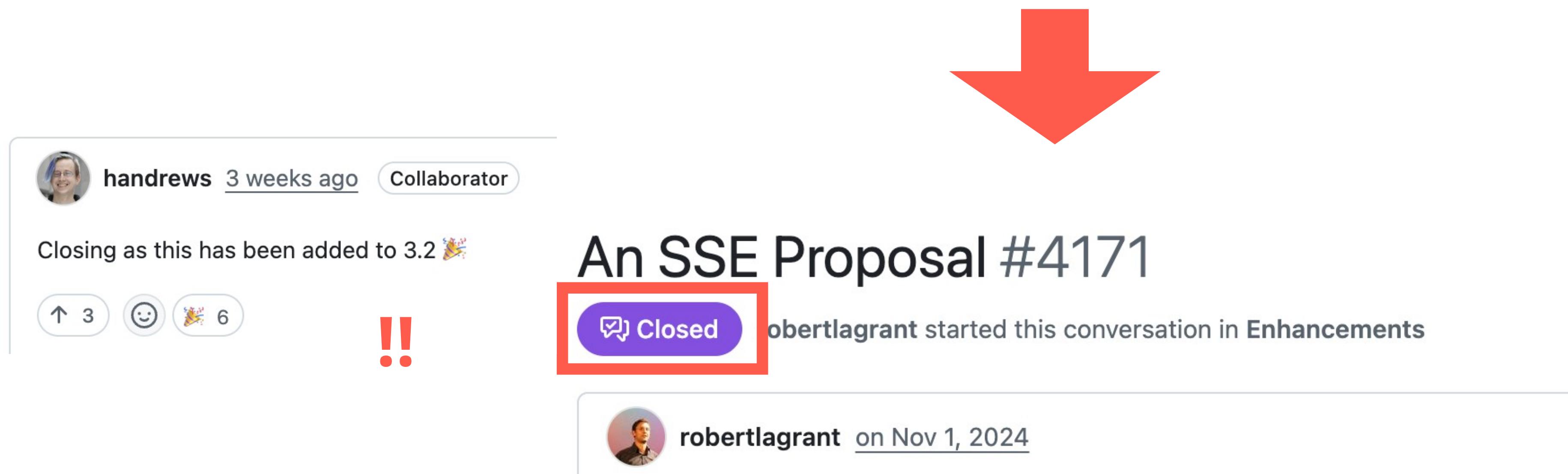
?

Closed robertlagrant started this conversation in Enhancements

 robertlagrant on Nov 1, 2024

<https://github.com/OAI/OpenAPI-Specification/discussions/4171>

- ▶ OpenAPI と SSE の相性があまり良くない
 - ▶ OpenAPI が RESTful な API 定義を記述することを主眼としているため Streaming な SSE との相性があまり良くない
 - ▶ クライアントの自動生成は今ある一般的なツールでは困難



handrews 3 weeks ago Collaborator

Closing as this has been added to 3.2 🎉

↑ 3 !! 6

An SSE Proposal #4171

Closed robertlagrant started this conversation in Enhancements

robertlagrant on Nov 1, 2024

<https://github.com/OAI/OpenAPI-Specification/discussions/4171>

ブロック色々 : OpenAPI 3

- ▶ OpenAPI 3.2 から **Sequential Media Types** が導入される
 - ▶ e.g., **application/json**, **text/event-stream**
- ▶ Sequential Media Types の各アイテムのスキーマを **itemSchema** で定義できるようになる

参照 : <https://spec.openapis.org/oas/v3.2.0.html>

```
content:  
  text/event-stream:  
    itemSchema:  
      $ref: "#/components/schemas/Event"  
      required: [event]  
      oneOf:  
        - properties:  
            event:  
              const: addString  
        - properties:  
            event:  
              const: addInt64  
            data:  
              $comment: |  
                Since the `data` field is a string,  
                we need a format to signal that it  
                should be handled as a 64-bit integer.  
              format: int64  
        - properties:  
            event:  
              const: addJson  
            data:  
              $comment: |  
                These content fields indicate  
                that the string value should  
                be parsed and validated as a  
                JSON document (since JSON is not  
                a binary format, `contentEncoding`  
                is not needed)  
            contentMediaType: application/json  
            contentSchema:  
              type: object  
              required: [foo]  
              properties:  
                foo:  
                  type: integer
```

ブロックカードではないが便利にしていく取り組み

async gem のフック : unhandled な例外をハンドルする

Add configurable custom exception handlers for unhandled errors #389

 Closed moznion wants to merge 1 commit into `socketry:main` from `moznion:unhandled_exception_handler` 



ioquatix commented on Jul 14

Member

...

Thanks for your contribution and sorry for not getting back to you sooner.

I'm basically against adding all the extra complexity.

However, I understand your problem and there is a hook that should allow you to achieve what you want with minimum effort.

[async/lib/async/task.rb](#)

Lines 382 to 384 in `ec3c870`

```
382  def warn(*args)
383    Console.warn(*args)
384  end
```

There are two options:

- Write your own `Async::Console` interface, the basic requirements are shown here:
<https://github.com/socketry/async/blob/ec3c8705cf519a3482e9504f34787c026a208ada/lib/async/console.rb>
- Replace `Async::Task#warn` with your own implementation.

Ruby is already a dynamic language, so we don't need to introduce even more dynamic dispatch. The above two interfaces are official extension points, although replacing `#warn` is probably a little more kludgy.

Is that sufficient for your needs?



<https://github.com/socketry/async/pull/389>

SSE 関連のテストを書きやすくするための rspec matcher

moznion/rspec-sse-matchers

A collection of RSpec matchers for testing Server-Sent Events (SSE)



2
Contributors

1
Issue

3
Stars

0
Forks



Example:

```
# In your controller
def index
  response.headers['Cache-Control'] = 'no-store'
  response.headers['Content-Type'] = 'text/event-stream'
  sse = SSE.new(response.stream)

  sse.write({id: 1, message: 'Hello'}, event: 'message', id: 1)
  sse.write({id: 2, message: 'World'}, event: 'update', id: 2)
  sse.close
end

# In your spec
RSpec.describe 'SSE endpoint', type: :request do
  it 'sends the expected events' do
    get '/events', headers: { 'Accept' => 'text/event-stream' }

    # Verify the response indicates a successful SSE connection
    expect(response).to be_sseSuccessfullyOpened

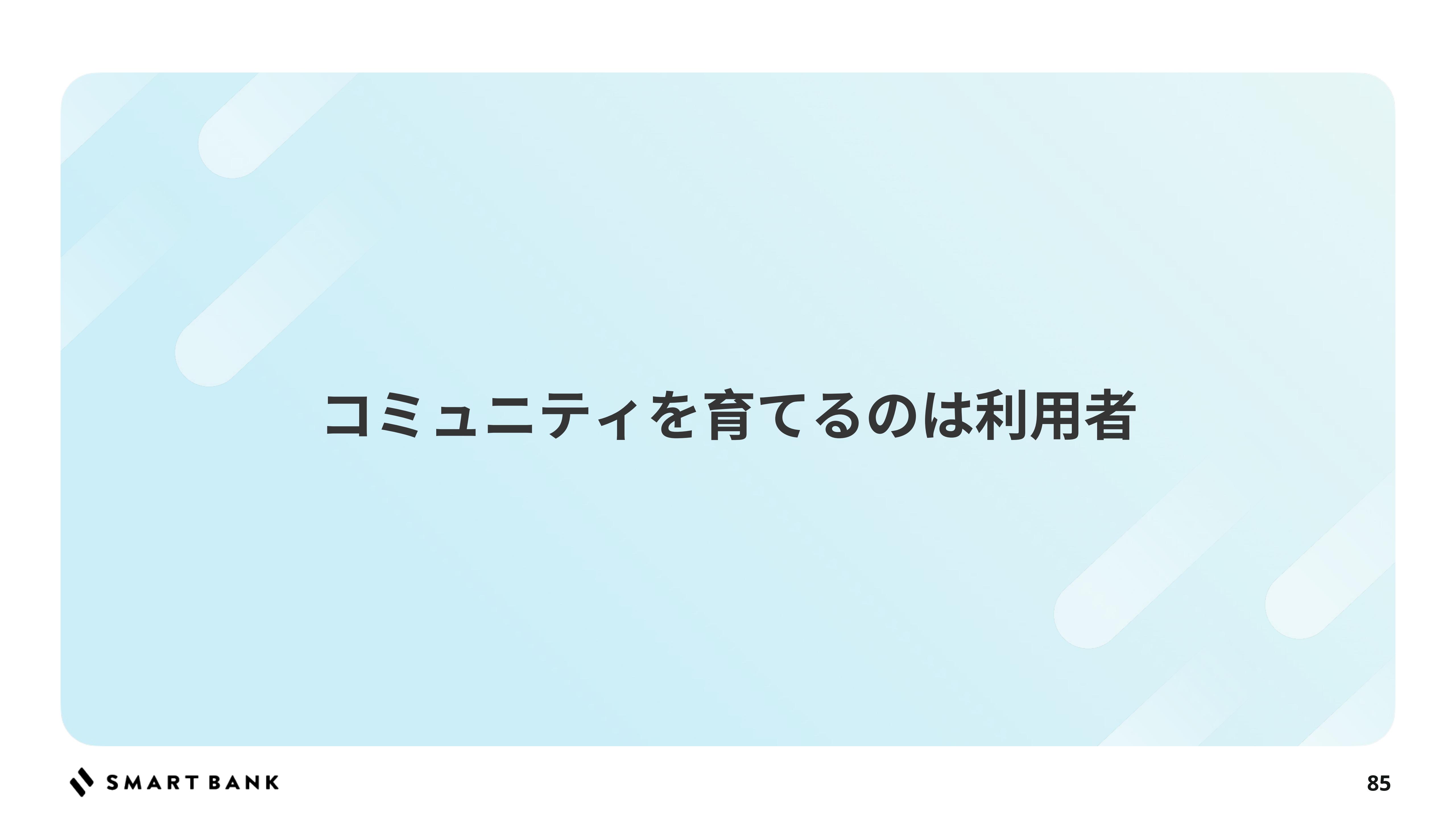
    # Verify the event types
    expect(response).to be_sseEventTypes(['message', 'update'])

    # Verify that the response is properly closed
    expect(response).to be_sseGracefullyClosed
  end
end
```

SSE 周りは今でも十分にサポートされている（ありがたい）
一方で使ってみるともっと便利にできる箇所があることに気付く

(SSE に限らず)

そのような事柄に出くわした時には
コミュニティに還元できると良いですね



コミュニティを育てるのは利用者

まとめ

- ▶ SSE について説明しました
- ▶ SSE の有効な用途について説明しました
- ▶ SSE と `async` を組み合わせた非同期処理による恩恵・実装パターンを説明しました
- ▶ エコシステムを便利にできるチャンスを見付けたらどしどし貢献していきましょう



Kaiqi on Rails 2025

参考情報

- ▶ **HTML Standard - 9.2 Server-sent events**
<https://html.spec.whatwg.org/multipage/server-sent-events.html>
- ▶ **Server-sent events - Web APIs | MDN**
https://developer.mozilla.org/en-US/docs/Web/API/Server-sent_events