Some more adventure of Happy Eyeballs 

塩井美咲 (@shioimm / @coe401_)

2024/08/31

RubyKaigi 2024 follow up

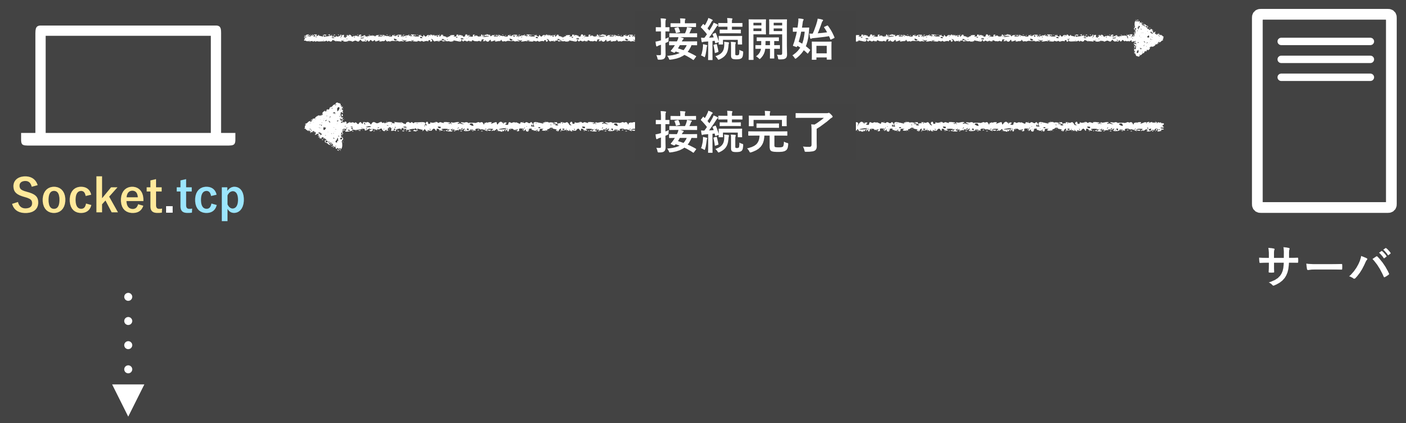
pp self

- 塩井美咲（しおい）
- @shioimm (GitHub)
- @coe401_ (Twitter?) / @coe401 (Bluesky)

RubyKaigi 2024で
「An adventure of Happy Eyeballs」という
発表をしました

これまでのあらすじ

あるところにSocket.tcpというメソッドが

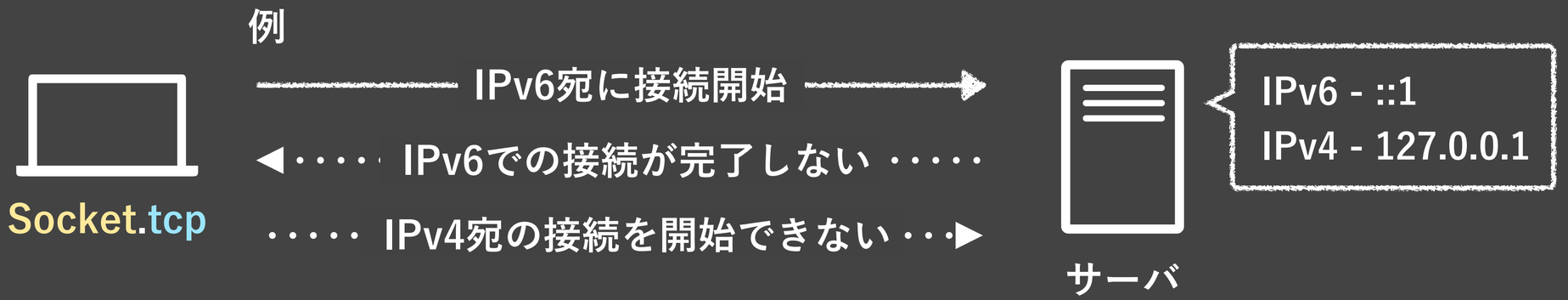


指定のサーバーに対して、TCPで接続したクライアントソケットを返す

（ソケット…コンピュータにとっての通信の出入り口）

これまでのあらすじ

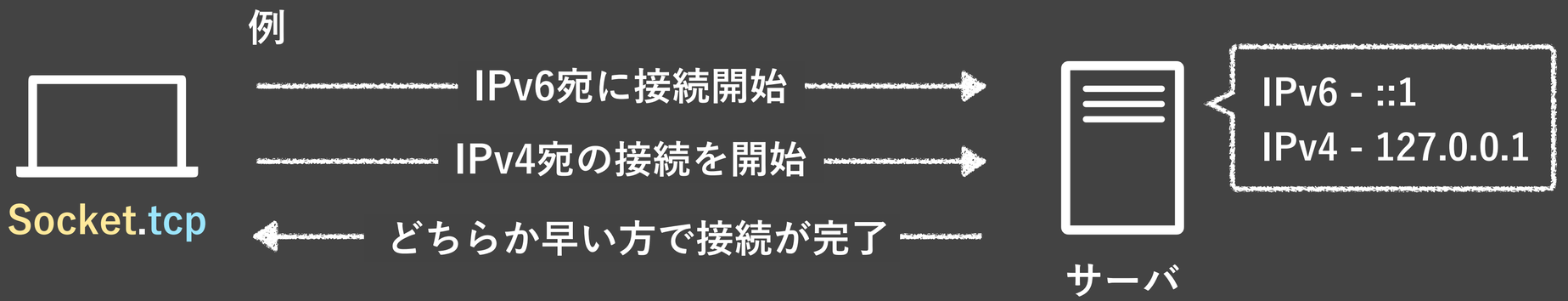
接続先サーバがIPv6 / IPv4アドレス両方を持つ場合、
従来のSocket.tcpは名前解決や接続試行を「直列に」行う



➡ 名前解決や接続を先に行おうとした
アドレスファミリの接続性が悪いと、
後続の処理が実行できないという課題があった

これまでのあらすじ

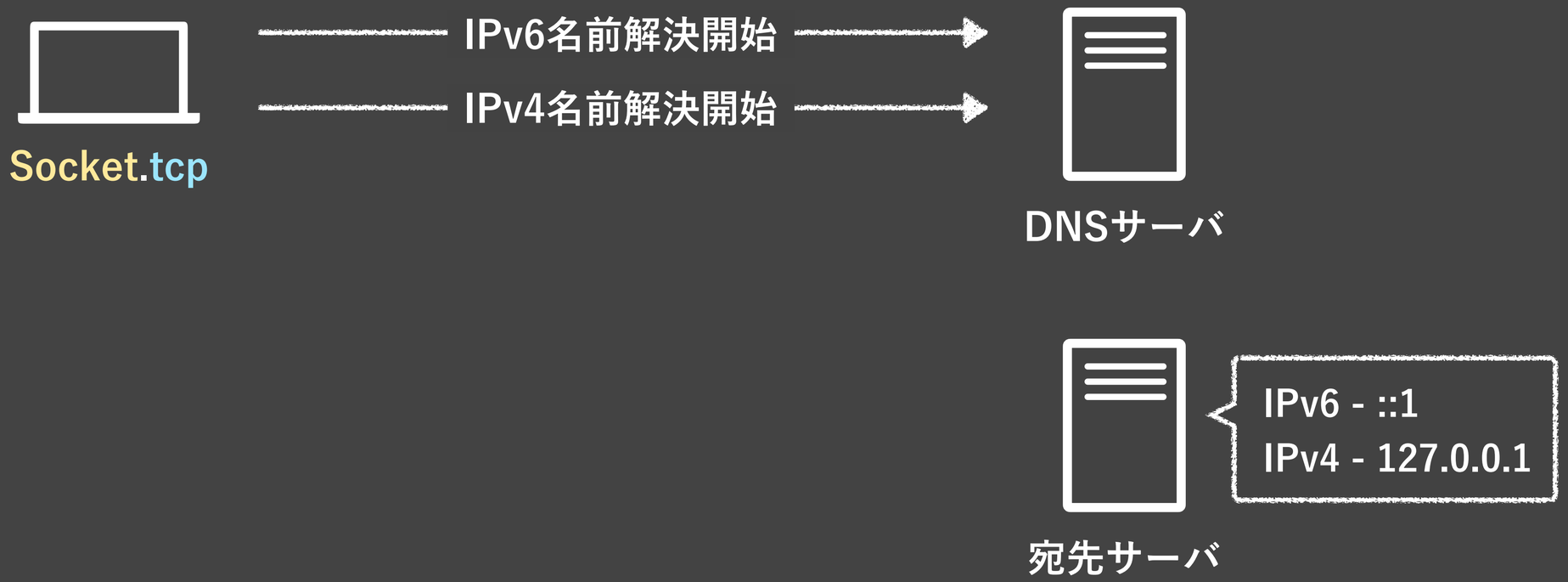
RFC8305にて、この課題を解決するためのアルゴリズム、
Happy Eyeballs Version 2 (HEv2) が規定されている



HEv2は名前解決や接続試行を
「一つずつ直列に」ではなく「複数並行に」行うことで、
効率よく接続できるようにする、というもの

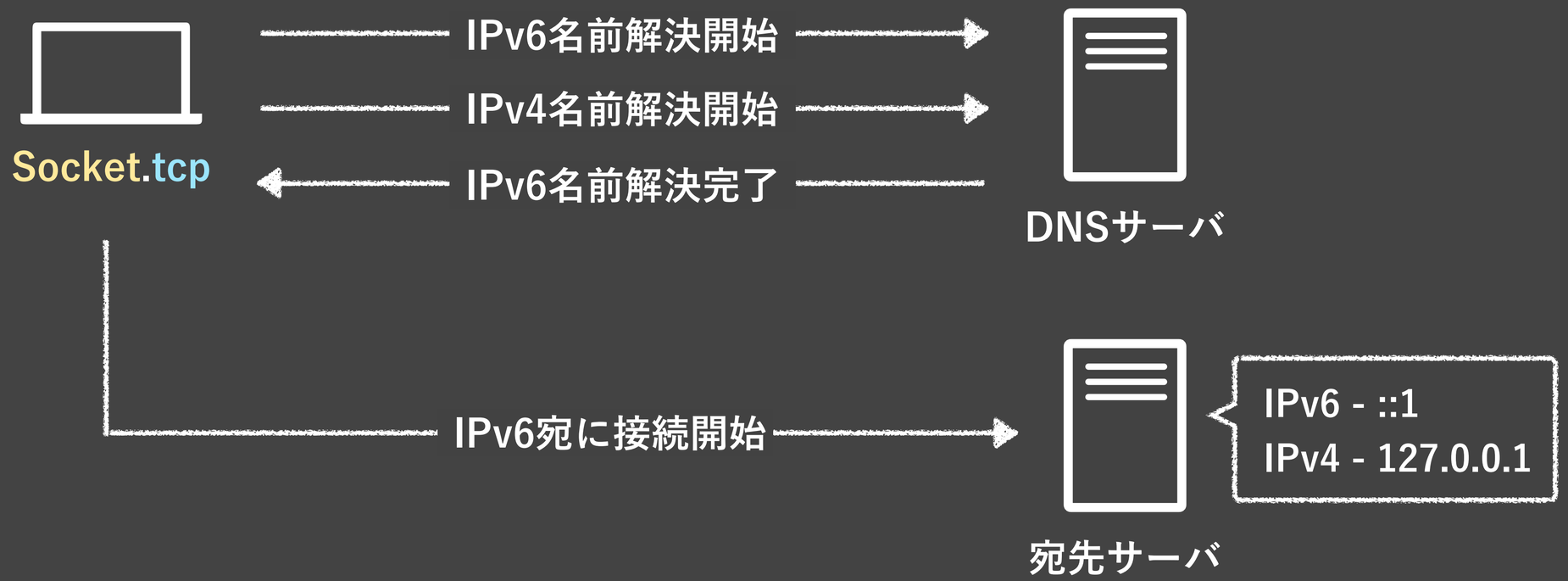
これまでのあらすじ: 1分でわかるHEv2

IPv6とIPv4の名前解決を並行して両方同時に開始する



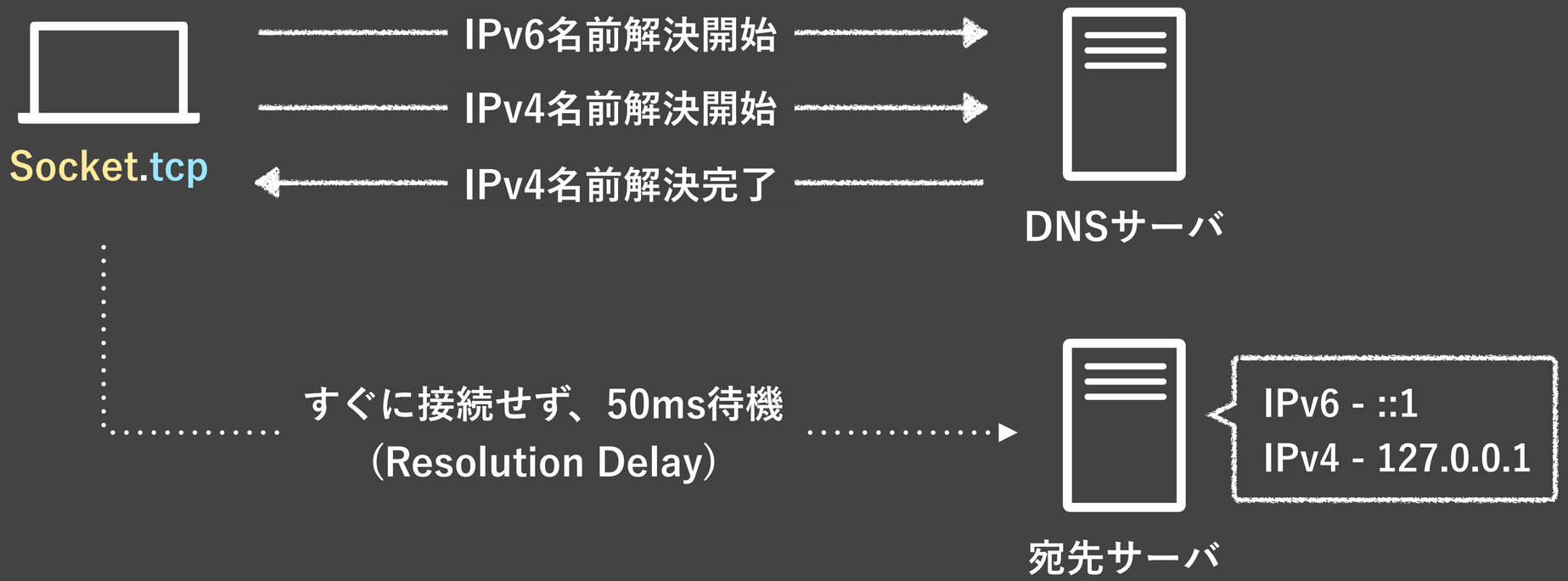
これまでのあらすじ: 1分でわかるHEv2

先に解決できたアドレス宛に接続を開始する



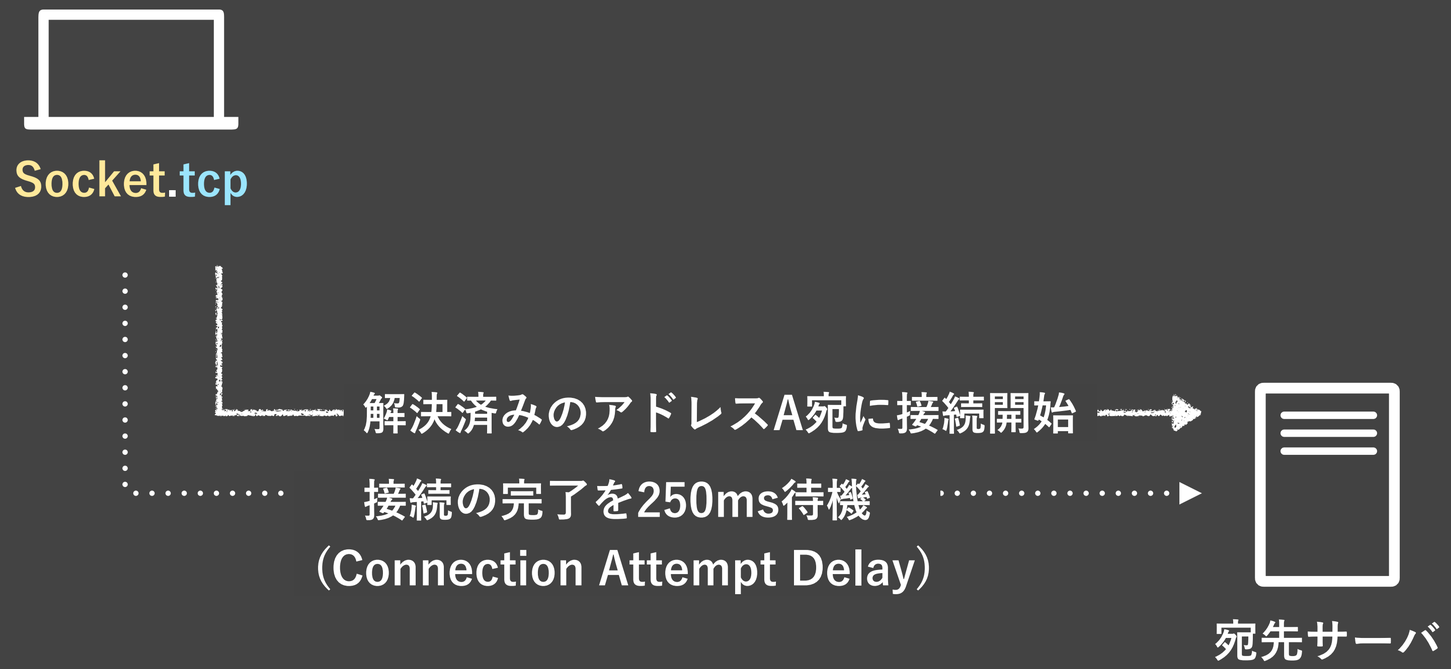
これまでのあらすじ: 1分でわかるHEv2

先にIPv4の名前解決が完了した場合即座に接続を開始せず
IPv6の名前解決を50ms待機する（Resolution Delay）
（HEv2ではIPv6での接続を優先したい狙いがあるため）



これまでのあらすじ: 1分でわかるHEv2

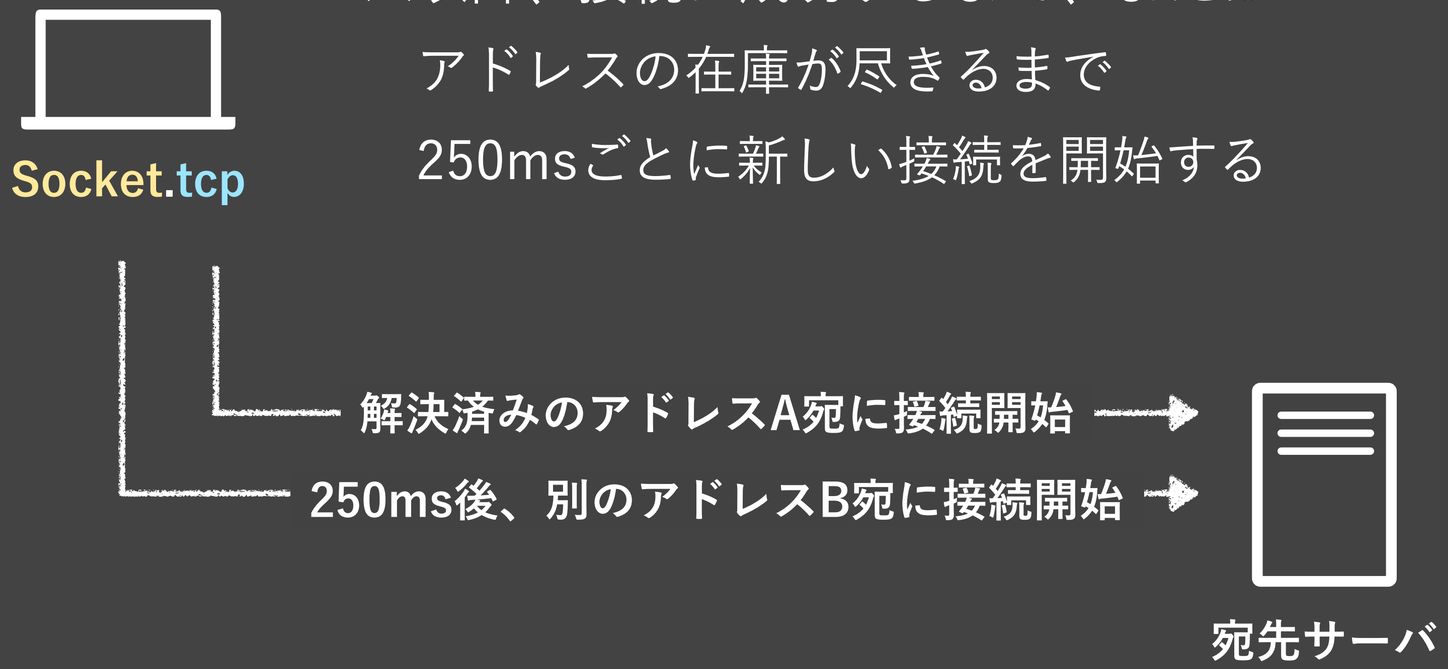
解決したアドレスのうち一つに対して接続を開始した後、
接続の完了を250ms待機する（Connection Attempt Delay）



これまでのあらすじ: 1分でわかるHEv2

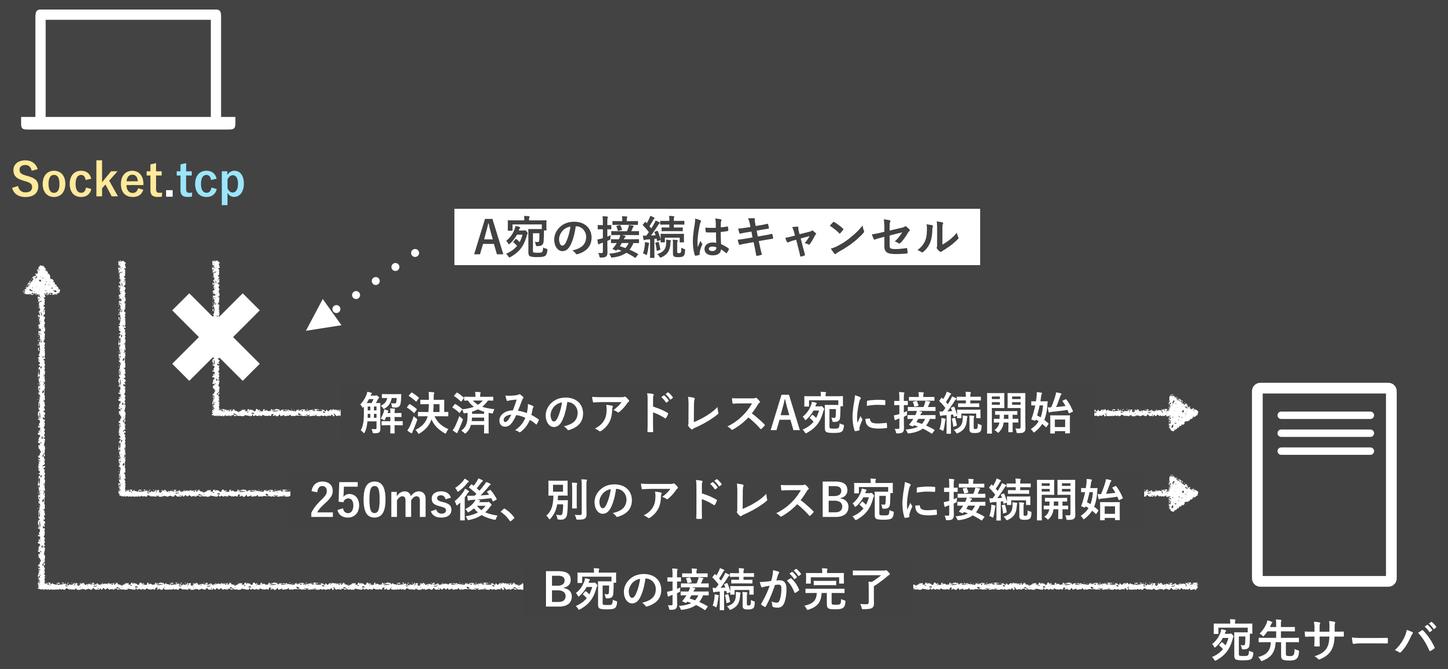
250ms以内に一つ目の接続が完了しなかった場合、
他の解決済みのアドレス宛に二つ目の接続を開始する

※以降、接続に成功するまで、または
アドレスの在庫が尽きるまで
250msごとに新しい接続を開始する



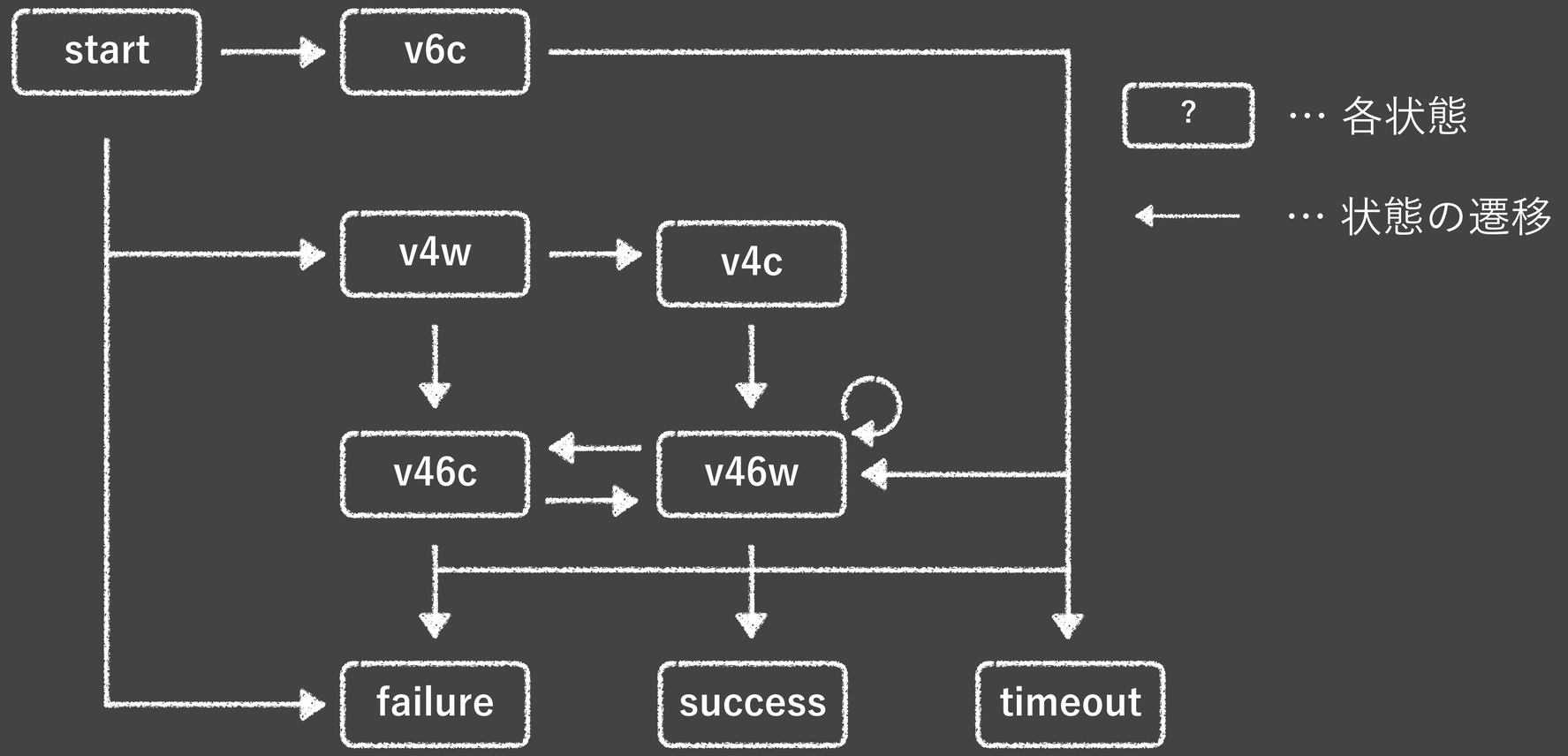
これまでのあらすじ: 1分でわかるHEv2

いずれかの接続が成功したら、成功した接続以外は
キャンセルする



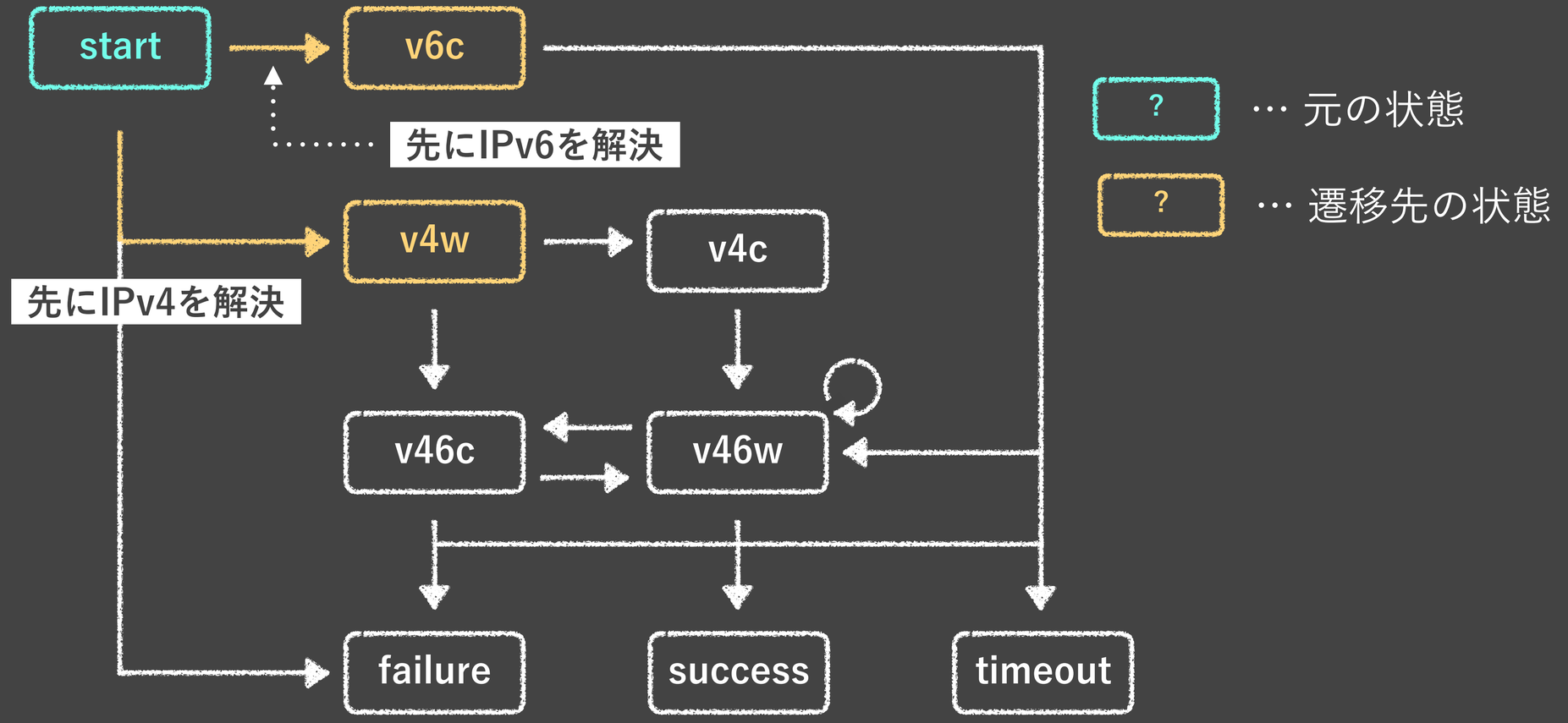
これまでのあらすじ: 状態遷移であらわすHEv2

HEv2の動作フローを表現する下記のような状態遷移図を作成し
これをベースにSocket.tcpのHEv2実装を行なった



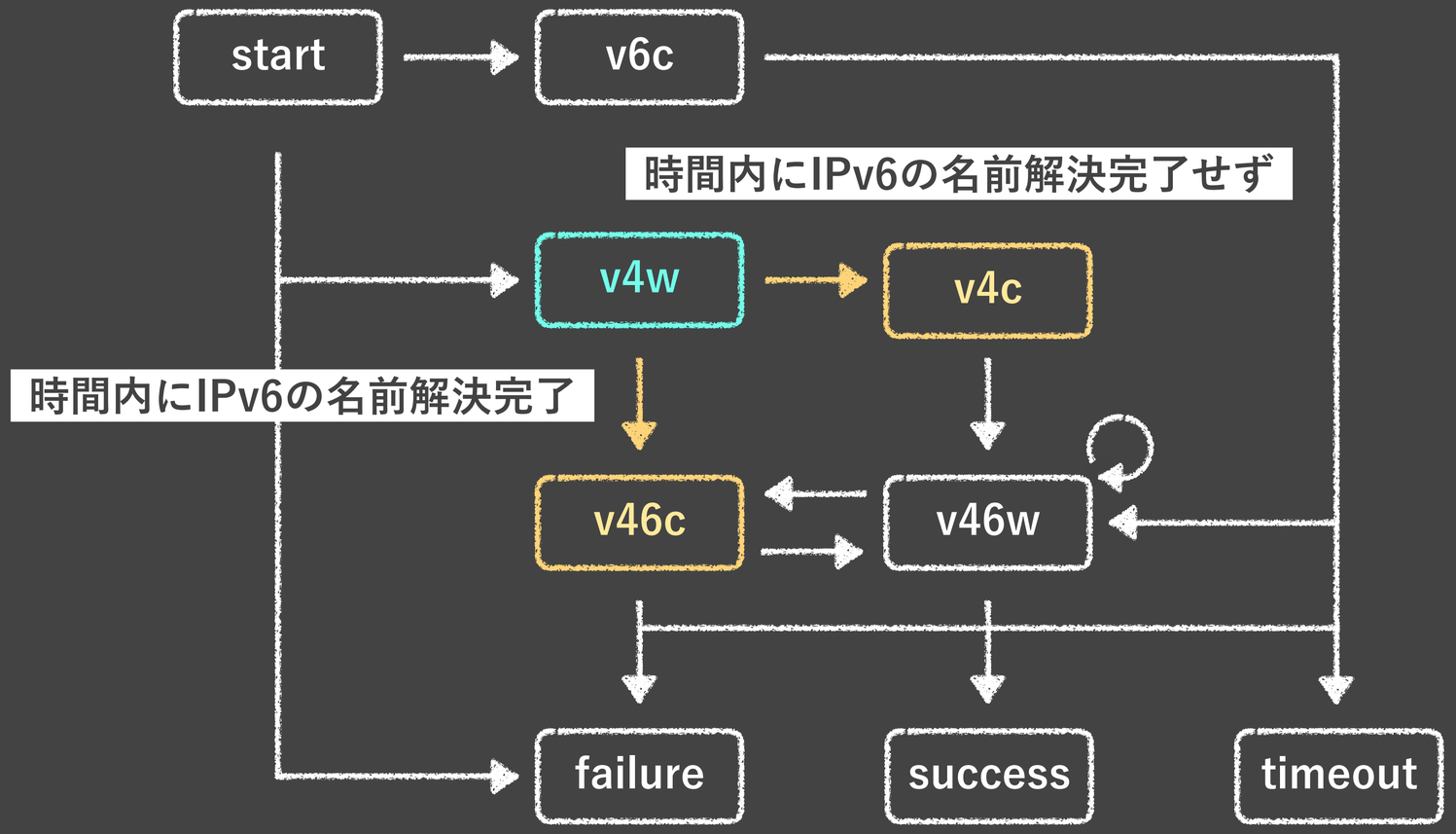
これまでのあらすじ: HEv2の状態遷移

start ... 開始時点での状態。IPv6とIPv4の名前解決を同時に開始する
どちらかが完了するまで待機する



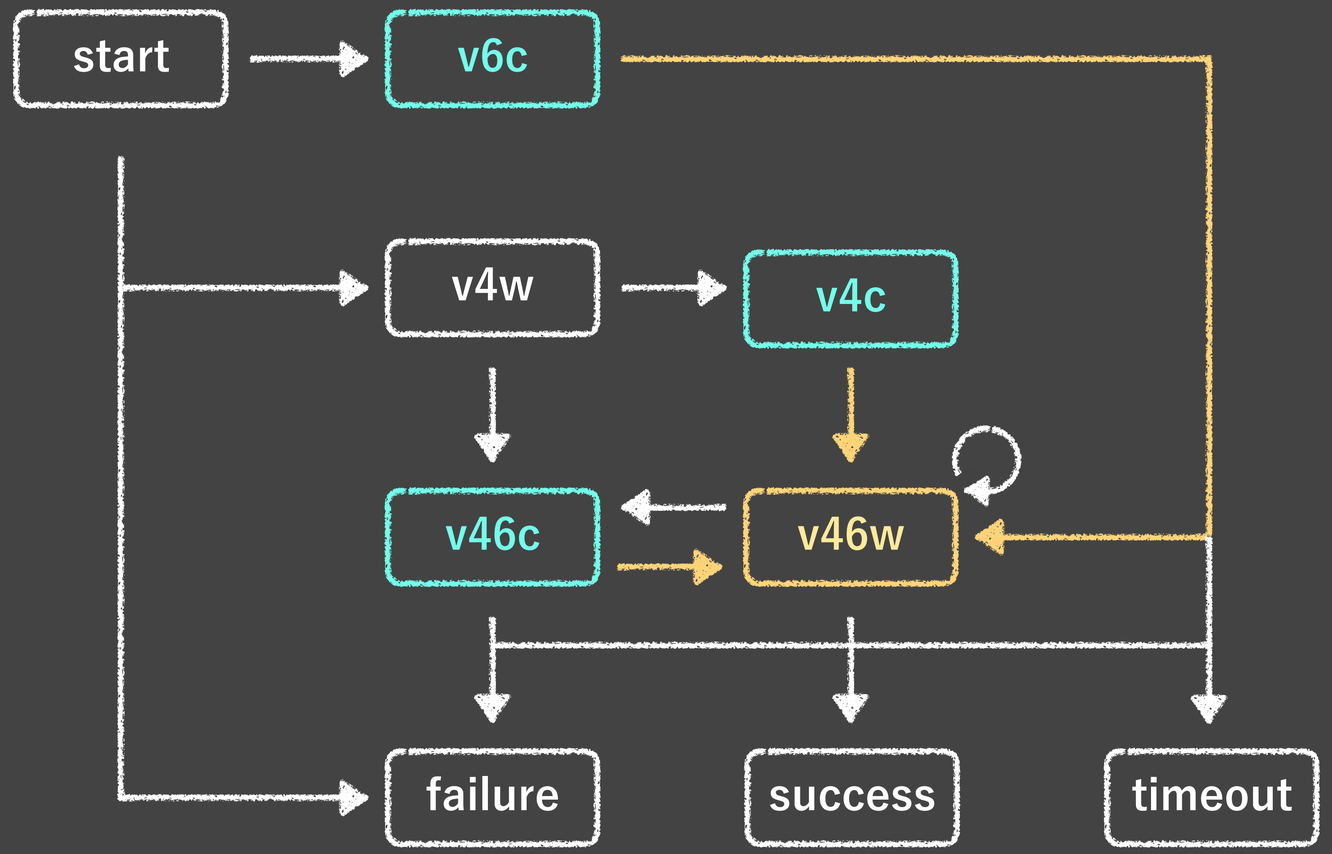
これまでのあらすじ: HEv2の状態遷移

v4w ... IPv6の名前解決を50ms待機 (Resolution Delay)



これまでのあらすじ: HEv2の状態遷移

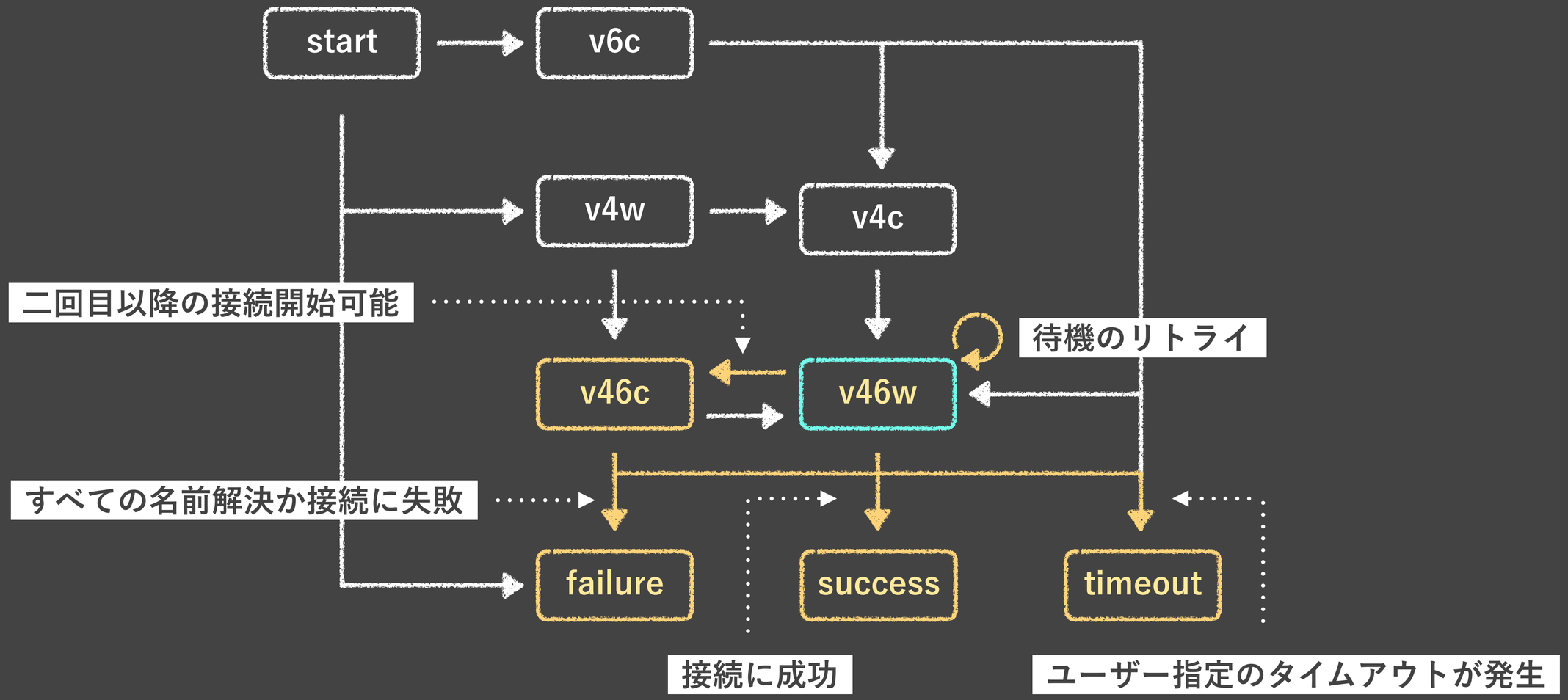
v6c ... IPv6アドレス宛に接続開始 / **v4c** ... IPv4アドレス宛に接続開始
v46c ... IPv4アドレスまたは IPv6アドレス宛に接続開始



※いずれの状態も接続開始後は**v46w**へ遷移して接続を待機

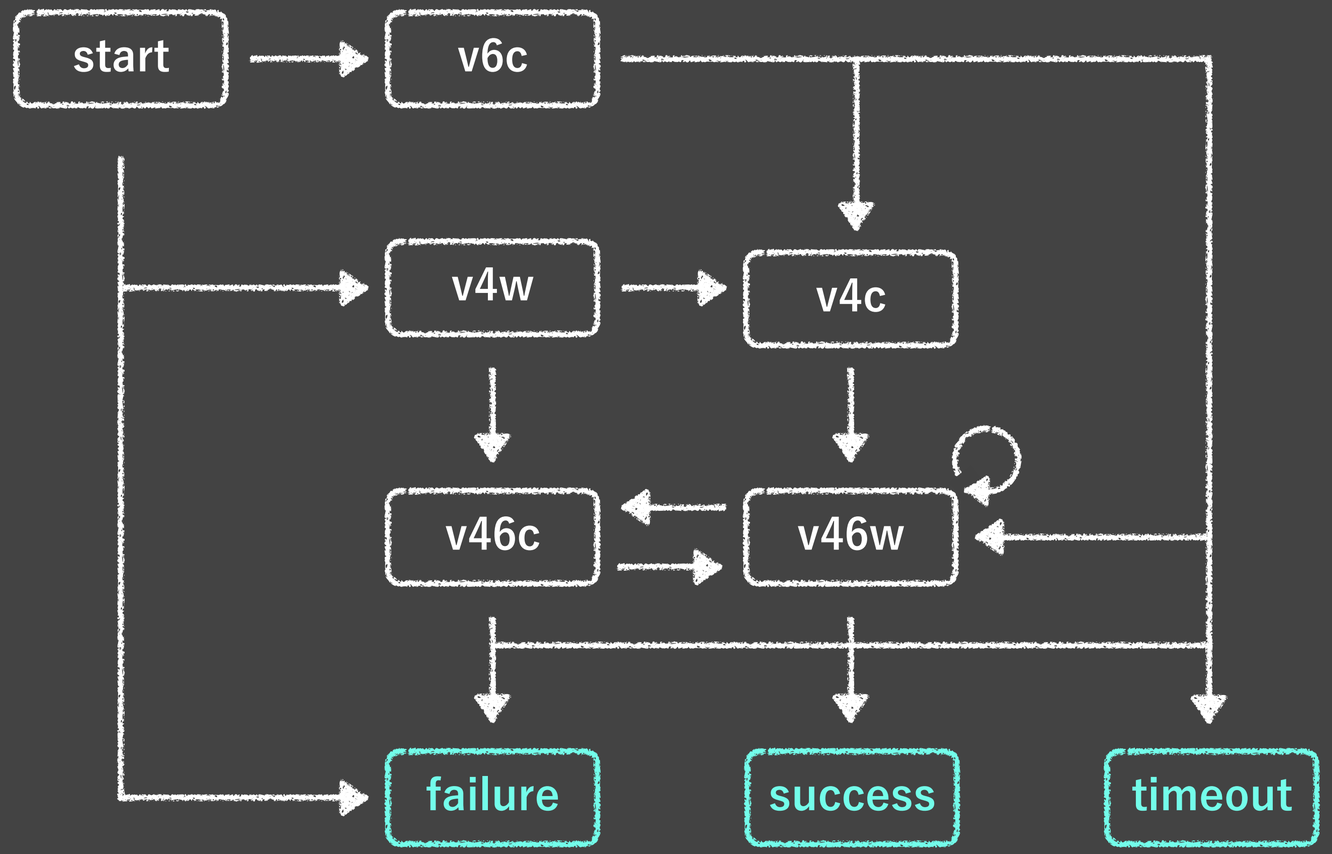
これまでのあらすじ: HEv2の状態遷移

v46w ... 接続の完了または 名前解決の完了を250ms待機
(Connection Attempt Delay)



これまでのあらすじ: HEv2の状態遷移

success / failure / timeout ... 終了状態



これまでのあらすじ: Socket.tcpの実装（全体像）

```
def self.tcp(host, port, ...)
  # ...
  state = :start

  loop do
    case state
    when :start                then  名前解決を開始する
    when :v4w                  then  IPv6の名前解決完了を50ms待機する
    when :v6c, :v4c, :v46c     then  接続を開始する
    when :v46w                 then  接続状態の確定と名前解決の完了を待機する
    when :success              then  接続に成功したソケットを返す
    when :failure              then  例外を発生させる
    when :timeout              then  タイムアウト例外を発生させる
    end
  end
end
```

実装にあたってはKernel#loopとcaseを
組み合わせることで先ほどの状態遷移を表現する

これまでのあらすじ: Socket.tcpの実装（全体像）

```
def self.tcp(host, port, ...)
  # ...
  state = :start

  loop do
    case state
    when :start                then  名前解決を開始する
    when :v4w                  then  IPv6の名前解決完了を50ms待機する
    when :v6c, :v4c, :v46c     then  接続を開始する
    when :v46w                 then  接続状態の確定と名前解決の完了を待機する
    when :success              then  接続に成功したソケットを返す
    when :failure              then  例外を発生させる
    when :timeout              then  タイムアウト例外を発生させる
    end
  end
end
```

state ... 「現在の状態」を管理する変数（初期状態としてstartを設定）

現在の状態

各状態で行う処理

ループ内では必ず「現在の状態」が設定されている。
状態に応じて行うべき処理を実行し、
次の状態を設定し、次のループに進む

これまでのあらすじ: Socket.tcpの実装（start）

```
case state
when :start（※名前解決を開始するための状態）
  # ...
  hostname_resolution_args = [host, port, hostname_resolution_queue]

  hostname_resolution_threads.concat(
    resolving_family_names.map { |family|
      thread_args = [family, *hostname_resolution_args]
      thread = Thread.new(*thread_args) { |*thread_args|
        hostname_resolution(*thread_args)
      }
      Thread.pass
      thread
    }
  )
  # ...
```

アドレスファミリごとにスレッドを作成

各子スレッドで名前解決メソッドを実行

start : アドレスファミリごとにスレッドを生成。
それぞれの子スレッドで名前解決用のメソッドを
呼び出すことにより、IPv6 / IPv4の名前解決を
並行に開始する。

これまでのあらすじ: Socket.tcpの実装（start）

```
case state
when :start（※名前解決を開始するための状態）
  # ...
  hostname_resolved, _, =
    IO.select(hostname_resolution_waiting, nil, nil, remaining)
  # ...
  family_name, res = hostname_resolution_queue.get
  # ...
  state = case family_name
          when :ipv6 then :v6c
          when :ipv4 then :v4w
          end
  selectable_addrinfos.add(family_name, res)

  # ...
  next
```

ユーザー指定のタイムアウト値
指定がない場合はnil = 無期限

どちらかの名前解決が完了するまで処理をブロック

名前解決の結果を取得

次の状態を決定

次の状態へ

start : IO.selectを用いて名前解決が完了するまで
(基本的には)無期限で待機する

これまでのあらすじ: Socket.tcpの実装（v4w）

```
case state
when :v4w （※先にIPv4を解決できた場合にIPv6の名前解決を待機するための状態）
  ipv6_resolved, _, =
    IO.select(hostname_resolution_waiting, nil, nil, RESOLUTION_DELAY)

  if ipv6_resolved
    family_name, res = hostname_resolution_queue.get
    selectable_addrinfos.add(family_name, res) unless res.is_a? Exception
    state = :v46c
  else
    state = :v4c
  end

  next
```

50ms

IPv6の名前解決が完了するかタイムアウトするまで処理をブロック

次の状態を決定

次の状態へ

v4w : IO.selectを用いてIPv6の名前解決を50ms待機。
待機時間内にIPv6の名前解決が完了したかによって、
次の状態を判断する。

これまでのあらすじ: Socket.tcpの実装 (v6c / v4c / v46c)

```
case state
when :v6c, :v4c, v46c （※接続を開始するための状態）
  # ...
  addrinfo = selectable_addrinfos.get ← 取得済みのアドレスを一つ取得
  # ...

  socket = Socket.new(addrinfo.pfamily, addrinfo.socktype, addrinfo.protocol)
  # ...

  result = socket.connect_nonblock(addrinfo, exception: false) ← 接続を開始
  # ...
  connecting_sockets.add(socket, addrinfo) ← 接続に使用したソケットを保存
  state = :v46w ← 次の状態をセット
  # ...

  next
    ↖ 次の状態へ
```

v6c / v4c / v46c : 解決済みのアドレスをひとつ取得し、
ノンブロッキングモードで接続を開始。
接続しに要したソケットを保存して v46w へ遷移

これまでのあらすじ: Socket.tcpの実装（v46w）

```
case state
when :v46w（※接続状態の確定と名前解決の完了を同時に待機するための状態）
  # ...
  hostname_resolved, connectable_sockets, =
    IO.select(hostname_resolution_waiting, connecting_sockets.all, nil, remaining)
  # ...

  if connectable_sockets&.any?  （接続状態が確定）
    # ...
  elsif hostname_resolved&.any?（名前解決が完了）
    # ...
  else（Connection Attempt Delayがタイムアウト）
    # ...
  end
  # ...

  next
```

250ms

接続状態が確定するか、名前解決が完了するか、
タイムアウトするまで処理をブロック

次の状態を決定

次の状態へ

v46w : IO.selectを用いて接続状態の確定か
名前解決の完了を250ms待機。IO.selectの結果
発生したイベントなどによって、次の状態を判断する

これまでのあらすじ: Socket.tcpの実装（終了状態）

```
case state
when :success （※接続に成功したソケットを返すための状態）
  break connected_socket
when :failure （※すべての名前解決と接続に失敗し、例外を発生させるための状態）
  raise last_error
when :timeout （※ユーザー指定のタイムアウトが発生し、例外を発生させるための状態）
  raise Errno::ETIMEDOUT, "user specified timeout"
end
```

success ➡ 接続に成功したソケットを返す
failure ➡ 最後に補足したエラーで例外を発生させる
timeout ➡ タイムアウトを表す例外を発生させる

これまでのあらすじ: Socket.tcpの実装

```
def self.tcp(host, port, ...)
  # ...
  state = :start
  loop do
    case state
    when :start                then  名前解決を開始する
    when :v4w                  then  IPv6の名前解決完了を待機する
    when :v6c, :v4c, :v46c     then  接続を開始する
    when :v46w                 then  接続状態の確定と名前解決の完了を待機する
    when :success              then  接続に成功したソケットを返す
    when :failure              then  例外を発生させる
    when :timeout              then  タイムアウト例外を発生させる
    end
  end
 end
```

完成!

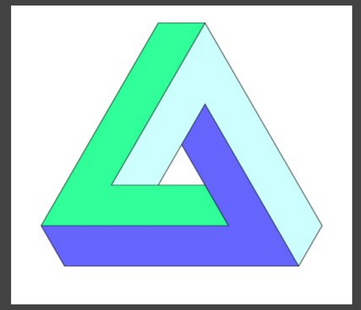
めでたしめでたし

~~めでたしめでたし~~

ではなかった

ではなかった

akrさん



実装を状態遷移に寄せすぎでは?

「実装を状態遷移に寄せすぎ」の例: start

```
case state
when :start (※名前解決を開始するための状態)
  # ...
  hostname_resolved, _, =
    IO.select(hostname_resolution_waiting, nil, nil, remaining)
  # ...

  family_name, res = hostname_resolution_queue.get
  # ...
  state = case family_name
          when :ipv6 then :v6c
          when :ipv4 then :v4w
          end
  selectable_addrinfos.add(family_name, res)

  # ...
  next
```

名前解決の結果を取得

先に解決できたアドレスファミリによって次の状態を決定

start : 先に名前解決が完了したのがIPv6だったのか、
IPv4だったのかによって、次の状態を判断している

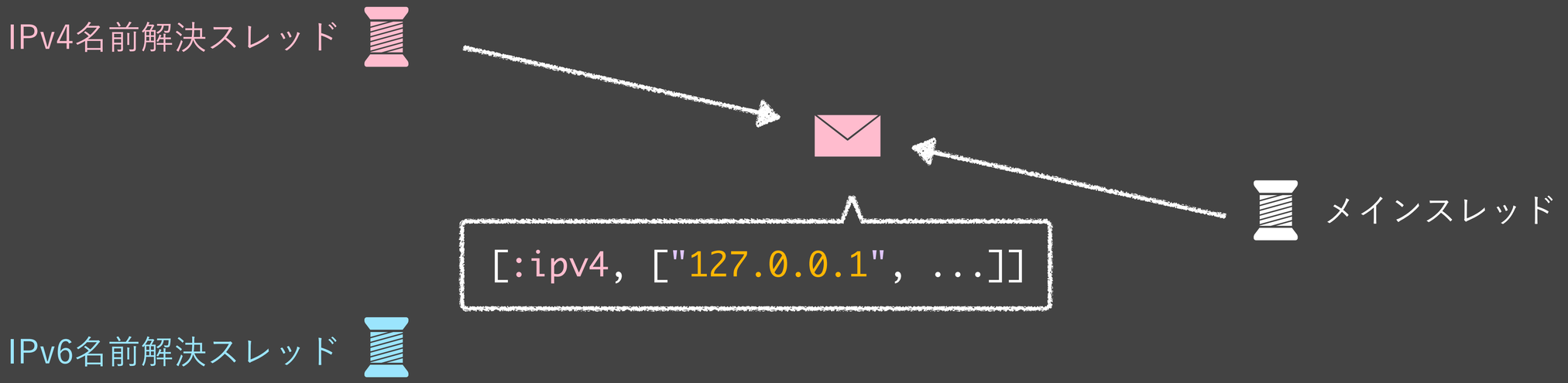
「実装を状態遷移に寄せすぎ」の例: start

```
case state
when :start (※名前解決を開始するための状態)
  # ...
  hostname_resolved, _, =
    IO.select(hostname_resolution_waiting, nil, nil, remaining)
  # ...

  family_name, res = hostname_resolution_queue.get
  # ...
  state = case family_name ← ???
          when :ipv6 then :v6c
          when :ipv4 then :v4w
          end
  selectable_addrinfos.add(family_name, res)

  # ...
  next
```

この時点で IPv6 / IPv4 ともに
名前解決が完了していたらどうなる?

スレッド間でのアドレスの受け渡しの仕組み

※「子スレッドが名前解決し、メインスレッドがそれを待機する」という処理の裏側



子スレッドは名前解決が完了した後、その結果として
解決したアドレスファミリ名と取得したアドレスを
要素に持つ配列をメインスレッドに受け渡す

スレッド間でのアドレスの受け渡しの仕組み

IPv4名前解決スレッド

共有キュー

メインスレッド

IPv6名前解決スレッド

受け渡しは、メインスレッド - 子スレッド間で
共有しているキューを介して行われる

スレッド間でのアドレスの受け渡しの仕組み

IPv4名前解決中…

IPv4名前解決スレッド

IPv6名前解決中…

IPv6名前解決スレッド

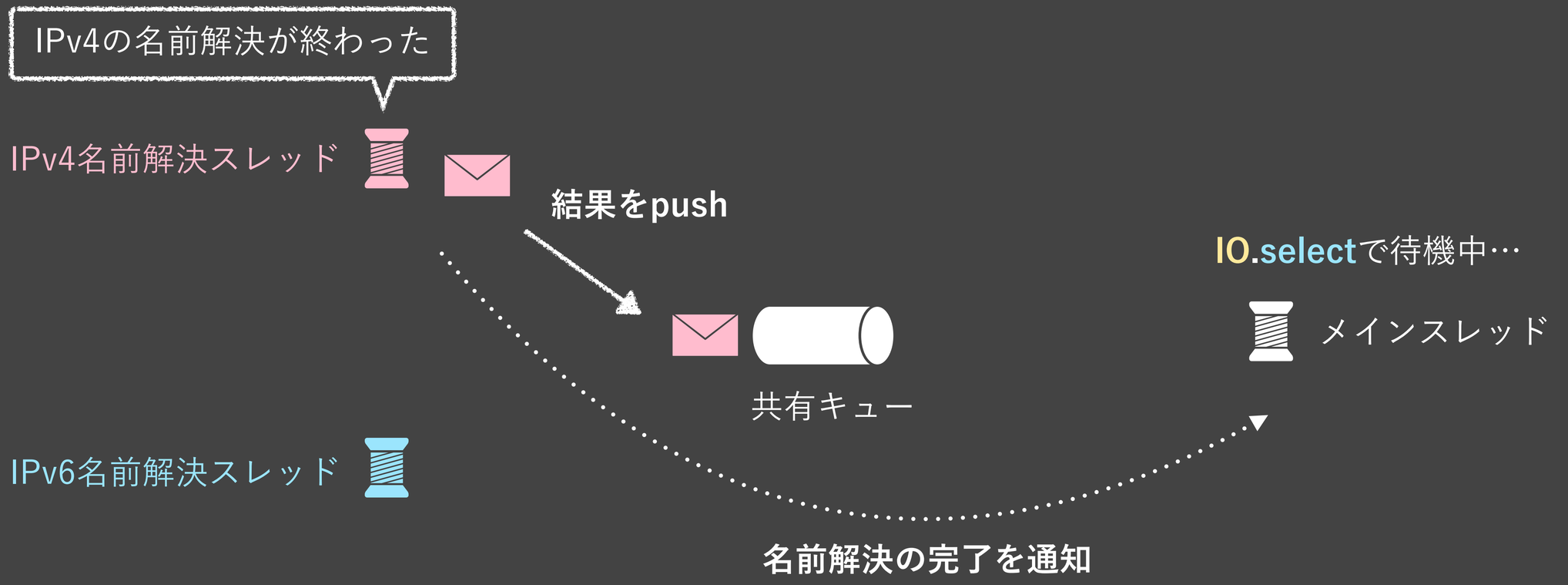
共有キュー

IO.selectで待機中…

メインスレッド

メインスレッドはIO.selectで名前解決を待機する

スレッド間でのアドレスの受け渡しの仕組み

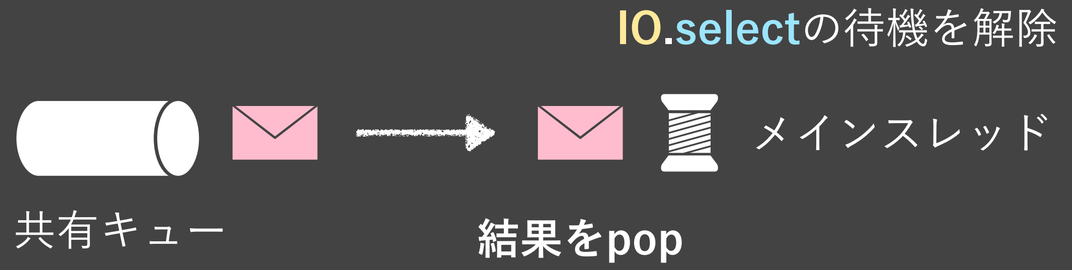


子スレッドは名前解決が終わり次第、
結果をキューにpushする
また、メインスレッドのIO.selectに完了を通知する

スレッド間でのアドレスの受け渡しの仕組み

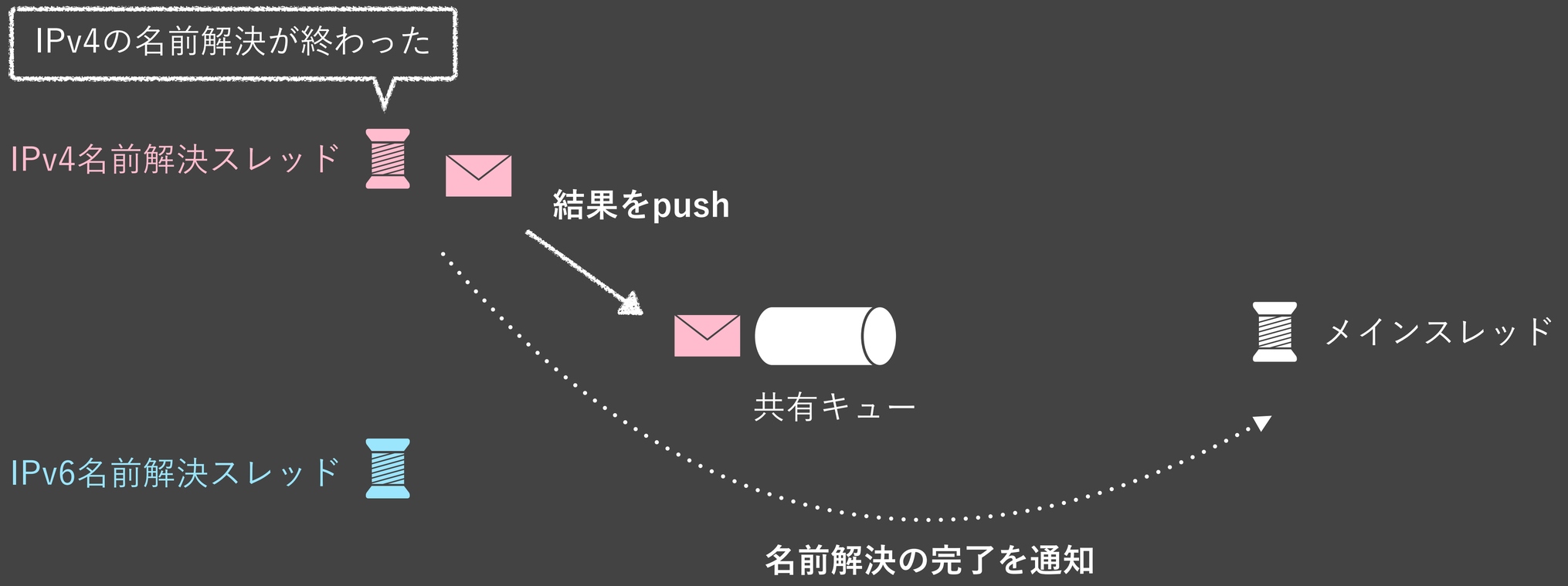
IPv4名前解決スレッド

IPv6名前解決スレッド



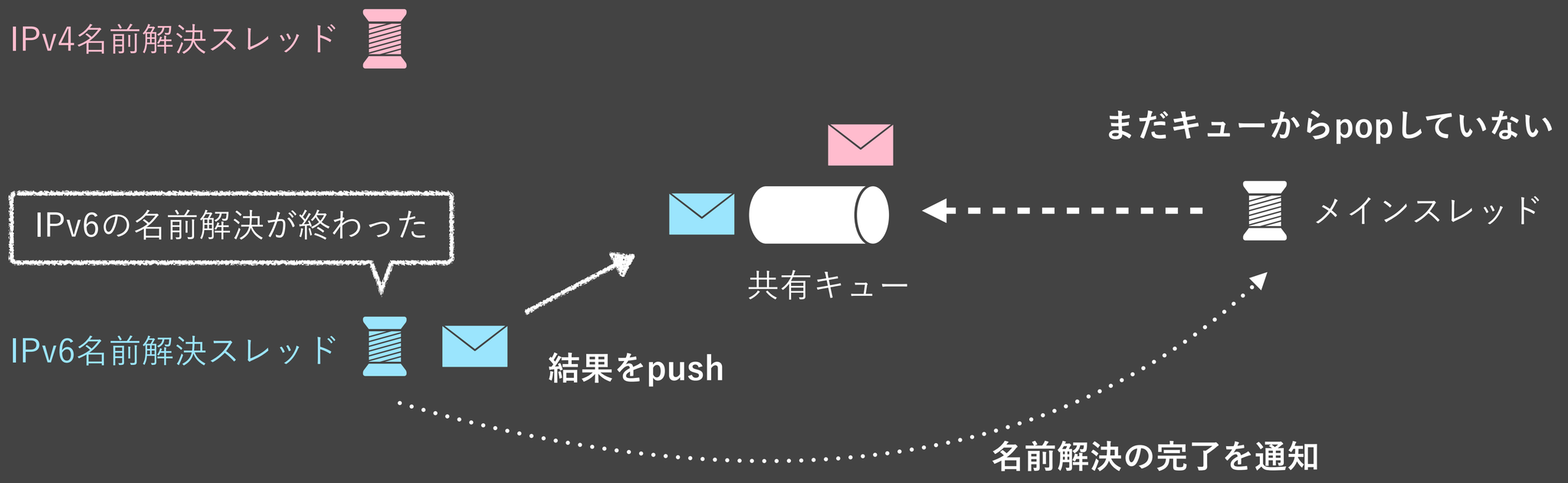
メインスレッドはIO.selectに通知を受け取ると、
待機を解除してキューから結果をpopする

スレッド間でのアドレスの受け渡しの仕組み



(例)
先に片方の子スレッドでIPv4の名前解決が完了して
キューに結果がpushされた後、

スレッド間でのアドレスの受け渡しの仕組み



メインスレッドがそれをpopする前に、
もう片方の子スレッドでIPv6の名前解決が完了し、
キューにIPv6の名前解決結果もpushされた場合

「実装を状態遷移に寄せすぎ」の例: start

```
case state
when :start (※名前解決を開始するための状態)
  # ...
  hostname_resolved, _, =
    IO.select(hostname_resolution_waiting, nil, nil, remaining)
  # ...
  family_name, res = hostname_resolution_queue.get
  # ...
  state = case family_name
          when :ipv6 then :v6c
          when :ipv4 then :v4w
          end
  selectable_addrinfos.add(family_name, res)

  # ...
  next
```

キューの先頭を取得

取得したアドレスファミリがIPv4なので、
次の状態をv4wと判断してしまう

メインスレッドはIO.selectの待機を解除し、キューの先頭からひとつ取得する（先頭はIPv4の名前解決結果）
メインスレッドは取得できたアドレスファミリから次の状態を判断するため、この場合はv4wへ遷移する。

「実装を状態遷移に寄せすぎ」の例: v4w

```
case state
when :v4w （※先にIPv4を解決できた場合にIPv6の名前解決を待機するための状態）
  ipv6_resolved, _, =
    IO.select(hostname_resolution_waiting, nil, nil, RESOLUTION_DELAY)

  if ipv6_resolved
    family_name, res = hostname_resolution_queue.get
    selectable_addrinfos.add(family_name, res) unless res.is_a? Exception
    state = :v46c
  else
    state = :v4c
  end

  next
```

すでにIPv6は解決済みなのに、不要にIO.selectを呼び出している!

キューの先頭を取得

v4wにて、IPv6はすでに解決済みであるにも関わらず
IPv6の名前解決を待機するためにIO.selectを呼び出す
（不要なIO.selectの呼び出しが発生している）

どうしてこうなった

```
case state
when :start (※名前解決を開始するための状態)
  # ...
  hostname_resolved, _, =
    IO.select(hostname_resolution_waiting, nil, nil, remaining)
  # ...
  family_name, res = hostname_resolution_queue.get
  # ...
  state = case family_name
          when :ipv6 then :v6c
          when :ipv4 then :v4w
          end


  # ...
  next
```

一つの状態からは必ず一つの状態へ遷移する（「状態の重ね合わせ」が発生する可能性が考慮されていない）

状態遷移を見た目のまま実装した結果、一つの状態からは必ず一つの状態へ遷移するという前提の実装になっている。一方、実際にはこの例のように「状態の重ね合わせ」が発生しうる。（**start**から**v6c**に遷移するための条件と**v4w**に遷移するための条件が同時に満たされうる）

問題点の整理

現在の実装の前提

- 実行中は「現在の状態」を管理する
- 「状態」ごとに行うべき処理が決まっている
- ーループの中で一つの「状態」を表現する
  （その「状態」で行うべき処理を行う。
  次の「状態」をセットして次のループに入る）

問題点の整理

現在の実装の問題

🗺️ 「状態の重ね合わせ」が発生した場合、
重なった状態 **a** / **b** のうちひとつ **a** を選択して遷移する

問題点の整理

現在の実装の問題

🗺️ 「状態の重ね合わせ」が発生した場合、
重なった状態 **a** / **b** のうちひとつ **a** を選択して遷移する

➡ 次のループでは状態 **a** で行うべき処理だけが実行され、
選択されなかった状態 **b** で行う処理が残る

問題点の整理

現在の実装の問題

🗺️ 「状態の重ね合わせ」が発生した場合、
重なった状態 **a** / **b** のうちひとつ **a** を選択して遷移する

➡ 次のループでは状態 **a** で行うべき処理だけが実行され、
選択されなかった状態 **b** で行う処理が残る

➡ 結果、誤った状態に遷移したり、
その先で不要な`IO.select`の呼び出しが発生する

問題点の整理

現在の実装の問題

「状態の重ね合わせ」が発生した場合、
重なった状態 **a** / **b** のうちひとつ **a** を選択して遷移する

➡ 次のループでは状態 **a** で行うべき処理だけが実行され、
選択されなかった状態 **b** で行う処理が残る

➡ 結果、誤った状態に遷移したり、
その先で不要な`IO.select`の呼び出しが発生する

実装が状態遷移に寄り過ぎている

元の実装

Introduction of Happy Eyeballs Version 2 (RFC8305) in Socket.tcp #9374

https://github.com/ruby/ruby/pull/9374

元の実装

Introduction of Happy Eyeballs Version 2 (RFC8305)
in Socket.tcp #9374

https://github.com/ruby/ruby/pull/9374

いったん無し

1から再実装

Improve Socket.tcp #11187

https://github.com/ruby/ruby/pull/11187

HEv2対応Socket.tcp(改)

元の実装コンセプト（再掲）

- 実行中は「現在の状態」を管理する
- 「状態」ごとに行うべき処理が決まっている
- 一ループの中で一つの「状態」を表現する

■ **課題**

一回のループの中で一つの状態しか表現できない

➡ そのループの中でまだ行うことができる処理が残っていても
それを実行することができない

HEv2対応Socket.tcp(改)

改善後の実装コンセプト

- 「現在の状態」を管理しない
- ループの中で行うことができる処理はすべて行う

HEv2対応Socket.tcp(改)の概観

```
def self.tcp(host, port, ...)
  タイムアウト変数の初期化
  名前解決の開始
  loop do

    if 接続開始の条件を満たしている then 接続を開始

    IO.select(<名前解決>, <接続>, nil, <タイムアウト値>)

    if 接続状態が確定したソケットがある then 接続確認　(成功していたらreturn)

    if 名前解決が完了 then 解決済みのIPアドレスを保存

    if 次のループに入ることができない then 例外を発生させる

  end
end
```

HEv2対応Socket.tcp(改)の概観

```
def self.tcp(host, port, ...)
  タイムアウト変数の初期化
  名前解決の開始
  loop do

    if 接続開始の条件を満たしている then 接続を開始

    IO.select(<名前解決>, <接続>, nil, <タイムアウト値>)

    if 接続状態が確定したソケットがある then 接続確認　(成功していたらreturn)

    if 名前解決が完了 then 解決済みのIPアドレスを保存

    if 次のループに入ることができない then 例外を発生させる

  end
end
```

状態管理をやめたことによりループ内のcaseが消え
代わりにif文による条件分岐が追加されている

ループのたびに上から順に実行される

HEv2対応Socket.tcp(改)の実装: ループに入る前

```
def self.tcp(host, port, ...)
  # ...
  resolution_delay_expires_at = nil
  connection_attempt_delay_expires_at = nil

  # ...
  loop do

    # ...

  end
end
```

タイムアウトを管理する変数

resolution_delay_expires_at
... IPv6の名前解決を50ms待機するための時間 (Resolution Delay) を格納する変数

connection_attempt_delay_expires_at
... 次の接続の開始まで250ms待機するための時間 (Connection Attempt Delay) を格納する変数

ループを開始

ループに入る前 ①
新たにタイムアウトを管理する変数を用意。
それぞれに初期値として**nil**をセットする

HEv2対応Socket.tcp(改)の実装: ループに入る前

```
def self.tcp(host, port, ...)
  # ...
  resolution_delay_expires_at = nil
  connection_attempt_delay_expires_at = nil

  # ...
  loop do

    # ...

  end
end
```

タイムアウトを管理する変数

※これらの変数にはそれぞれ、
・タイムアウトを待つ処理が発生した際に「いつまで待機するか」の時間がセットされる
・セットした時間が現在時刻よりも過去時間になったり待機が不要になるとnilがセットされる

ループを開始

HEv2対応Socket.tcp(改)の実装: ループに入る前

```
def self.tcp(host, port, ...)
  # ...
  hostname_resolution_threads.concat(
    resolving_family_names.map { |family|
      th_args = [family, host, port, hostname_resolution_result]
      thread = Thread.new(*th_args) { |*th_args | resolve_hostname(*th_args) }
      Thread.pass
      thread
    }
  )

  # ...

  loop do

    # ...

  end
end
```

アドレスファミリごとにスレッドを生成して名前解決を開始

ループを開始

ループに入る前 ② 名前解決を並行に開始する
(変更前はループの中 (**start**) で行なっていたが、
実際には全体で一回しか行わない処理のためここに移動)

HEv2対応Socket.tcp(改)の実装: 接続の開始

```
def self.tcp(host, port, ...)
  タイムアウト変数の初期化
  名前解決の開始
  # ...

  loop do ←  ループを開始


    # ...



  end
end
```

続いて、ループに入る

HEv2対応Socket.tcp(改)の実装: 接続の開始

```
loop do
  if resolution_store.any_addrinfos? &&
     !resolution_delay_expires_at &&
     !connection_attempt_delay_expires_at

    while (addrinfo = resolution_store.get_addrinfo)


      # 接続を開始するための処理


    end


    # ...


  end
end
```

ループを開始

接続開始の条件を満たしている?

ループの中ではまず
「接続開始の条件を満たしているかどうか」を確認する
- 解決済みのアドレスの在庫があること
- かつIPv6の名前解決を待機中ではないこと
- かつ接続開始の待機中ではないこと

HEv2対応Socket.tcp(改)の実装: 接続の開始

```
loop do
  # ...
  while (addrinfo = resolution_store.get_addrinfo)
    # ...
    socket = Socket.new(addrinfo.pfamily, addrinfo.socktype, addrinfo.protocol)
    socket.bind(local_addrinfo) if local_addrinfo
    result = socket.connect_nonblock(addrinfo, exception: false)

    # ...

    end
  end

  # ...

end
```

アドレスを取得

接続を開始

条件を満たす場合、解決済みのアドレスを一つ取得し、
ノンブロッキングモードで接続を開始

HEv2対応Socket.tcp(改)の実装: 接続の開始

```
loop do
  # ...
  while (addrinfo = resolution_store.get_addrinfo)
    # ...
    socket = Socket.new(addrinfo.pfamily, addrinfo.socktype, addrinfo.protocol)
    socket.bind(local_addrinfo) if local_addrinfo
    result = socket.connect_nonblock(addrinfo, exception: false)
    # ...
    if result == :wait_writable
      connection_attempt_delay_expires_at = now + CONNECTION_ATTEMPT_DELAY
      # ...
      connecting_sockets[socket] = addrinfo
      # ...
    end
  end

  # ...

end
```

現在時刻の250ms後をセット

connection_attempt_delay_expires_atに
次の接続開始までのタイムアウト時間をセットする

HEv2対応Socket.tcp(改)の実装: 接続の開始

```
loop do
  # ...
  while (addrinfo = resolution_store.get_addrinfo)
    # ...
    socket = Socket.new(addrinfo.pfamily, addrinfo.socktype, addrinfo.protocol)
    socket.bind(local_addrinfo) if local_addrinfo
    result = socket.connect_nonblock(addrinfo, exception: false)
    # ...
    if result == :wait_writable
      connection_attempt_delay_expires_at = now + CONNECTION_ATTEMPT_DELAY
      # ...
      connecting_sockets[socket] = addrinfo
      # ...
    end
  end

  # ...

end
```

接続に使用したソケットを保存

接続に使ったソケットを保存したら、このif文を抜ける

HEv2対応Socket.tcp(改)の実装: 名前解決と接続の待機

```
loop do
  if 接続開始の条件を満たしている then 接続を開始
  # ...

  hostname_resolved, writable_sockets, except_sockets = IO.select(
    hostname_resolution_notifier,
    connecting_sockets.keys,
    is_windows_environment ? connecting_sockets.keys : nil,
    second_to_timeout(current_clock_time, ends_at),
  )

  if 接続状態が確定したソケットがある then 接続確認  (成功していたらreturn)
  if 名前解決が完了 then 解決済みのIPアドレスを保存

  # ...

end
```

接続状態の確定または名前解決の完了を待機

続いて、IO.selectを用いて接続状態の確定か
名前解決の完了を待機する
(変更前のSocket.tcpのv46wで行っていた処理と同じ)

HEv2対応Socket.tcp(改)の実装: 接続状態の確認

```
loop do
  IO.select(<名前解決>, <接続>, nil, <タイムアウト値>)
  # ...
  if writable_sockets&.any?
    while (writable_socket = writable_sockets.pop)
      is_connected = is_windows_environment || (
        sockopt = writable_socket.getsockopt(Socket::SOL_SOCKET, Socket::SO_ERROR)
        sockopt.int.zero?
      )
      if is_connected
        connecting_sockets.delete writable_socket
        return writable_socket
      else
        # ...
      end
    end
  end
  # ...
end
```

接続状態が確定したソケットがある?

IO.selectが待機を解除したら、実行結果として
「接続状態が確定したソケットがあるかどうか」を確認

HEv2対応Socket.tcp(改)の実装: 接続状態の確認

```
loop do
  IO.select(<名前解決>, <接続>, nil, <タイムアウト値>)
  # ...
  if writable_sockets&.any?
    while (writable_socket = writable_sockets.pop)
      is_connected = is_windows_environment || (
        sockopt = writable_socket.getsockopt(Socket::SOL_SOCKET, Socket::SO_ERROR)
        sockopt.int.zero?
      )
      if is_connected
        connecting_sockets.delete writable_socket
        return writable_socket
      else
        # ...
      end
    end
  end
  # ...
end
```

接続状態の確認

接続に成功している?

接続できたソケットを返す

条件が満たされている場合は、対象のソケットの
接続状態を確認し、適切な処理を行う

HEv2対応Socket.tcp(改)の実装: 名前解決の完了

```
loop do
  IO.select(<名前解決>, <接続>, nil, <タイムアウト値>)
  # ...
  if hostname_resolved&.any?
    while (family_and_result = hostname_resolution_result.get)
      family_name, result = family_and_result

      if result.is_a? Exception
        # ...
      else
        resolution_store.add_resolved(family_name, result)
      end

      # ...

    end
  end
  # ...
end
```

名前解決が完了した?

さらにIO.selectの実行結果として
「名前解決が完了しているかどうか」を確認

HEv2対応Socket.tcp(改)の実装: 名前解決の完了

```
loop do
  IO.select(<名前解決>, <接続>, nil, <タイムアウト値>)
  # ...
  if hostname_resolved&.any?
    while (family_and_result = hostname_resolution_result.get)
      family_name, result = family_and_result

      if result.is_a? Exception
        # ...
      else
        resolution_store.add_resolved(family_name, result)
      end

      # ...

    end
  end
  # ...
end
```

アドレスを取得できる限り取得

取得できたアドレスを保存

条件が満たされていれば、解決したアドレスを
取得できるかぎり取得して保存する。

HEv2対応Socket.tcp(改)の実装: 名前解決の完了

```
loop do
  IO.select(<名前解決>, <接続>, nil, <タイムアウト値>)
  # ...
  if hostname_resolved&.any?
    while (family_and_result = hostname_resolution_result.get)
      # ...
      if resolution_store.resolved?(:ipv4)
        if resolution_store.resolved?(:ipv6)

          # ...

        elsif resolution_store.resolved_successfully?(:ipv4)
          resolution_delay_expires_at = now + RESOLUTION_DELAY
        end
      end
    end
  end
  # ...
end
```

IPv4の名前解決が完了した?

かつ、IPv6の名前解決が完了していない

現在時刻の50ms後をセット

IPv4の名前解決が完了しており、かつIPv6の名前解決が完了していない場合はresolution_delay_expires_atにIPv6の名前解決を待機する時間をセットする

HEv2対応Socket.tcp(改)の実装: 名前解決の完了

```
loop do
  IO.select(<名前解決>, <接続>, nil, <タイムアウト値>)
  # ...
  if hostname_resolved&.any?
    while (family_and_result = hostname_resolution_result.get)
      # ...
      if resolution_store.resolved?(:ipv4)
        if resolution_store.resolved?(:ipv6)
          hostname_resolution_notifier = nil
          resolution_delay_expires_at = nil
          user_specified_resolv_timeout_at = nil
        elsif resolution_store.resolved_successfully?(:ipv4)
          resolution_delay_expires_at = now + RESOLUTION_DELAY
        end
      end
    end
  end
  # ...
end
```

IPv4の名前解決が完了した?

かつ、IPv6の名前解決が完了した

nilをセット

IPv4 / IPv6ともに名前解決が完了している場合は
resolution_delay_expires_atに**nil**をセットする

HEv2対応Socket.tcp(改)の実装: 次のループへ

```
loop do
  if 接続状態が確定したソケットがある then 接続確認 (成功していたらreturn)
  if 名前解決が完了 then 解決済みのIPアドレスを保存
  # ...
  if resolution_store.empty_addrinfos?
    if connecting_sockets.empty? && resolution_store.resolved_all_families?
      raise last_error
    end

    if (expired?(now, user_specified_resolv_timeout_at) || ...) &&
       (expired?(now, user_specified_connect_timeout_at) || ...)
      raise Errno::ETIMEDOUT, 'user specified timeout'
    end
  end
end
```

次のループに入ることができる?

ループの最後に、
「次のループに入ることができるかどうか」確認

HEv2対応Socket.tcp(改)の実装: 次のループへ

```
loop do
  if 接続状態が確定したソケットがある then 接続確認 (成功していたらreturn)
  if 名前解決が完了 then 解決済みのIPアドレスを保存
  # ...

  if resolution_store.empty_addrinfos?
    if connecting_sockets.empty? && resolution_store.resolved_all_families?
      raise last_error
    end

    if (expired?(now, user_specified_resolv_timeout_at) || ...) &&
       (expired?(now, user_specified_connect_timeout_at) || ...)
      raise Errno::ETIMEDOUT, 'user specified timeout'
    end
  end
end
```

これ以上実行できる処理がない?

最後に補足したエラーで例外を発生させる

・解決済みのアドレスの在庫がない
・かつ、接続中のソケットがない
・かつ、すべての名前解決が終わっている 場合は
これ以上実行できる処理がないため、例外を発生させる

HEv2対応Socket.tcp(改)の実装: 次のループへ

```
loop do
  if 接続状態が確定したソケットがある then 接続確認 (成功していたらreturn)
  if 名前解決が完了 then 解決済みのIPアドレスを保存
  # ...

  if resolution_store.empty_addrinfos?
    if connecting_sockets.empty? && resolution_store.resolved_all_families?
      raise last_error
    end

    if (expired?(now, user_specified_resolv_timeout_at) || ...) &&
       (expired?(now, user_specified_connect_timeout_at) || ...)
      raise Errno::ETIMEDOUT, 'user specified timeout'
    end
  end
end
```

ユーザー指定のタイムアウトを超過済み?

タイムアウトを表す例外を発生させる

ユーザー指定のタイムアウトが超過している場合、
タイムアウトを表す例外を発生させる。

HEv2対応Socket.tcp(改)の実装: 次のループへ

```
loop do
  if 接続状態が確定したソケットがある then 接続確認 (成功していたらreturn)
  if 名前解決が完了 then 解決済みのIPアドレスを保存
  # ...

  if resolution_store.empty_addrinfos?
    if connecting_sockets.empty? && resolution_store.resolved_all_families?
      raise last_error
    end

    if (expired?(now, user_specified_resolv_timeout_at) || ...) &&
       (expired?(now, user_specified_connect_timeout_at) || ...)
      raise Errno::ETIMEDOUT, 'user specified timeout'
    end
  end
end
```

次のループへ

それ以外の場合は次のループに進む (To be continued...)

HEv2対応Socket.tcp(改)：残りの課題

```
def self.tcp(host, port, ...)
  タイムアウト変数の初期化
  名前解決の開始
  loop do

    if 接続開始の条件を満たしている then 接続を開始

    IO.select(<名前解決>, <接続>, nil, <タイムアウト値>)

    if 接続状態が確定したソケットがある then 接続確認　(成功していたらreturn)

    if 名前解決が完了 then 解決済みのIPアドレスを保存

    if 次のループに入ることができない then 例外を発生させる

  end
end
```

残りの課題：IO.selectの待機時間

```
loop do

  # ...

  hostname_resolved, writable_sockets, except_sockets = IO.select(
    hostname_resolution_notifier,
    connecting_sockets.keys,
    is_windows_environment ? connecting_sockets.keys : nil,
    second_to_timeout(current_clock_time, ends_at),
  )

  # ...

end
```

（例）
IPv4 / IPv6ともに名前解決中 ➡ 無期限
IPv4名前解決後、IPv6名前解決を待機する ➡ 50ms
接続を開始後、次の接続開始まで待機する ➡ 250ms

待機時間（ループのたびに変わる可能性がある…）

変更後のSocket.tcpでは、ループの中で一回だけIO.selectを呼び出す。このIO.selectで待機するべき時間は、そのときの状況によって異なる

残りの課題：IO.selectの待機時間

```
loop do
  # ...
  ends_at =
    if resolution_store.any_addrinfos?
      resolution_delay_expires_at || connection_attempt_delay_expires_at
    else
      [user_specified_resolv_timeout_at, user_specified_connect_timeout_at].compact.max
    end

  hostname_resolved, writable_sockets, except_sockets = IO.select(
    hostname_resolution_notifier,
    connecting_sockets.keys,
    is_windows_environment ? connecting_sockets.keys : nil,
    second_to_timeout(current_clock_time, ends_at),
  )

  # ...

end
```

「いつまで待機するか」を選択

IO.selectに渡すための適切な待機時間を選択する
ロジックを導入

「いつまで待機するか」を選択する

```
loop do
  # ...
  ends_at =
   if resolution_store.any_addrinfos?
```

← 解決済みのアドレスの在庫がある場合

「解決済みのアドレスの在庫がある」状況とは：

- **Resolution Delay中**
  （IPv4の名前解決が完了済み、かつ接続開始前にIPv6の名前解決を待機中）
- **Connection Attempt Delay中 あるいは超過済み**
  （すでに一つ以上接続を開始済み、かつ解決済みのアドレスの在庫がある）

...のどちらか

```
  # ...
end
```

「いつまで待機するか」を選択する

```
loop do
  # ...
  ends_at =
   if resolution_store.any_addrinfos?
     resolution_delay_expires_at
  # ...
end
```

解決済みのアドレスの在庫がある場合

Resolution Delay中：

変数resolution_delay_expires_atに
セットされているタイムアウト値を返り値とする

「いつまで待機するか」を選択する

```
loop do
  # ...
  ends_at =
   if resolution_store.any_addrinfos?
     resolution_delay_expires_at || connection_attempt_delay_expires_at

  # ...
end
```

解決済みのアドレスの在庫がある場合

Connection Attempt Delay中 あるいは超過済み：

変数connection_attempt_delay_expires_atに
セットされているタイムアウト値（あるいはnil）を返り値とする

「いつまで待機するか」を選択する

```
loop do
  # ...
  ends_at =
   if resolution_store.any_addrinfos?
     resolution_delay_expires_at || connection_attempt_delay_expires_at
   else
     nil
   end

  # ...
end
```

解決済みのアドレスの在庫がない場合

「解決済みのアドレスの在庫がない」状況とは：

- まだIPv6 / IPv4ともに名前解決が終わっていない状況
- 全ての解決済みのアドレス宛に接続を開始し、在庫がなくなった状況

...のどちらか

「いつまで待機するか」を選択する

```
loop do
  # ...
  ends_at =
   if resolution_store.any_addrinfos?
     resolution_delay_expires_at || connection_attempt_delay_expires_at
   else
     [user_specified_resolv_timeout_at, user_specified_connect_timeout_at].compact.max
   end

  # ...
end
```

解決済みのアドレスの在庫がない場合

いずれの場合も、ユーザー指定のタイムアウトを管理する変数に
格納されている値の大きい方を返す
ユーザー指定のタイムアウトが明示的に指定されていない場合、
これらの変数には**Float::INFINITY**が格納されている（= **無期限に待機**）

「いつまで待機するか」を選択する

```
loop do
  # ...
  ends_at =
   if resolution_store.any_addrinfos?
     resolution_delay_expires_at || connection_attempt_delay_expires_at
   else
     [user_specified_resolv_timeout_at, user_specified_connect_timeout_at].compact.max
   end

  hostname_resolved, writable_sockets, except_sockets = IO.select(
    hostname_resolution_notifier,
    connecting_sockets.keys,
    is_windows_environment ? connecting_sockets.keys : nil,
    second_to_timeout(current_clock_time, ends_at),
  )

  # ...

end
```

選択された待機時間を格納

実際の待機時間（ends_atから現在時刻を引いた差分）

「選択された待機時間 - 現在時刻」の差分を
実際の待機時間としてIO.selectに渡す

タイムアウトを管理する変数をリセット

```
loop do
  # ...
  hostname_resolved, writable_sockets, except_sockets = IO.select(
    hostname_resolution_notifier,
    connecting_sockets.keys,
    is_windows_environment ? connecting_sockets.keys : nil,
    second_to_timeout(current_clock_time, ends_at),
  )
  now = current_clock_time
  if expired?(now, resolution_delay_expires_at)
    resolution_delay_expires_at = nil
  end
  if expired?(now, connection_attempt_delay_expires_at)
    connection_attempt_delay_expires_at = nil
  end

  # ...

end
```

resolution_delay_expires_at が過去時間?

connection_attempt_delay_expires_at が過去時間?

IO.selectから返ってきた時点でタイムアウト時間が過去時間になっていた場合はタイムアウトの管理が不要になるのでそれぞれnilをセット

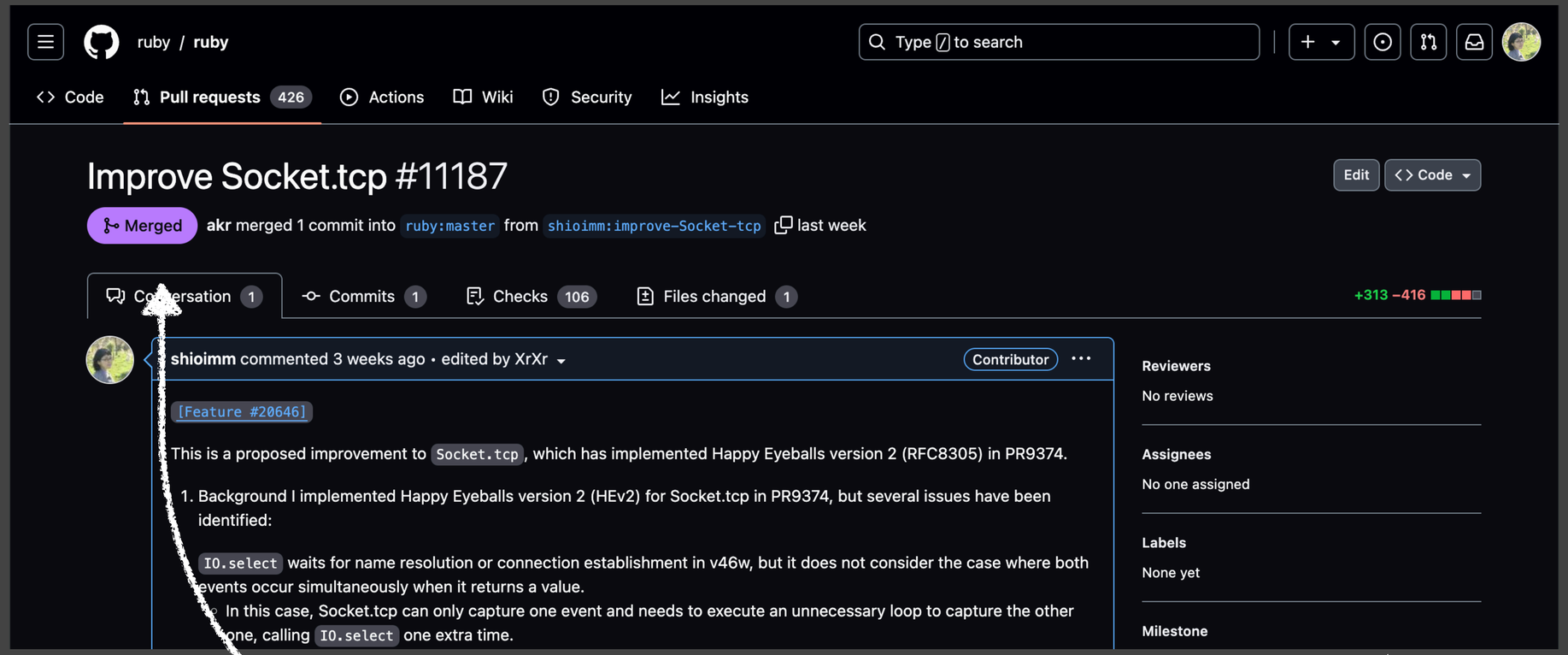
HEv2対応Socket.tcp(改)

```
def self.tcp(host, port, ...)
  タイムアウト変数の初期化
  名前解決の開始
  loop do

    if 接続開始の条件を満たしている then 接続を開始

    IO.select(<名前解決>, <接続>, nil, <タイムアウト値>)

    if 接続状態が確定したソケットがある then 接続確認　(成功していたらreturn)

    if 名前解決が完了 then 解決済みのIPアドレスを保存

    if 次のループに入ることができない then 例外を発生させる

  end
end
```

今度こそ完成!

Improve Socket.tcp #11187

https://github.com/ruby/ruby/pull/11187



Ruby 3.4のmasterにマージ済み

めでたしめでたし…?

TCPSocket.new 絶賛書き直し中

(状態遷移に寄せすぎバージョンで途中まで作り込んでいたため…)

https://speakerdeck.com/coe401_/an-adventure-of-happy-eyeballs?slide=52

To be continued...

ご清聴ありがとうございました

Special thanks to
@akr