ActionCable 与 AnyCable--原理及实现

— *by Mike Yang*

Mike Yang

滴滴出行

后端开发

Focusing

Mike Yang

yfractal

Edit profile

ShenZhen

yfractal@gmail.com

http://yfractal.github.io/

Organizations

参与开发的产品

主要使用的语言

Clojure, Erlang, Java

IM 需要应对的问题

高可用、高并发、低延迟

ActionCable

ActionCable

使服务端具有推送消息的能力

原有模式：客户端发起请求拉取信息。

WebSocket：服务端可以向客户端推送信息。

ActionCable 需要解决的问题

WebSocket 服务和 HTTP 服务职责划分

HTTP 服务不变，负责处理业务

WebSocket 服务负责维护链接和广播消息

架构由原来的单一的 APP Server，变为 APP Server + WebSocket Server

WebSocket 和 APP Server 服务间通信

APP Server -> WebSocket Server：消息队列

WebSocket Server -> APP Server：WebSocketServer 直接启动 APP 实例直接调用代码

长时间维护大量 WebSocket 链接

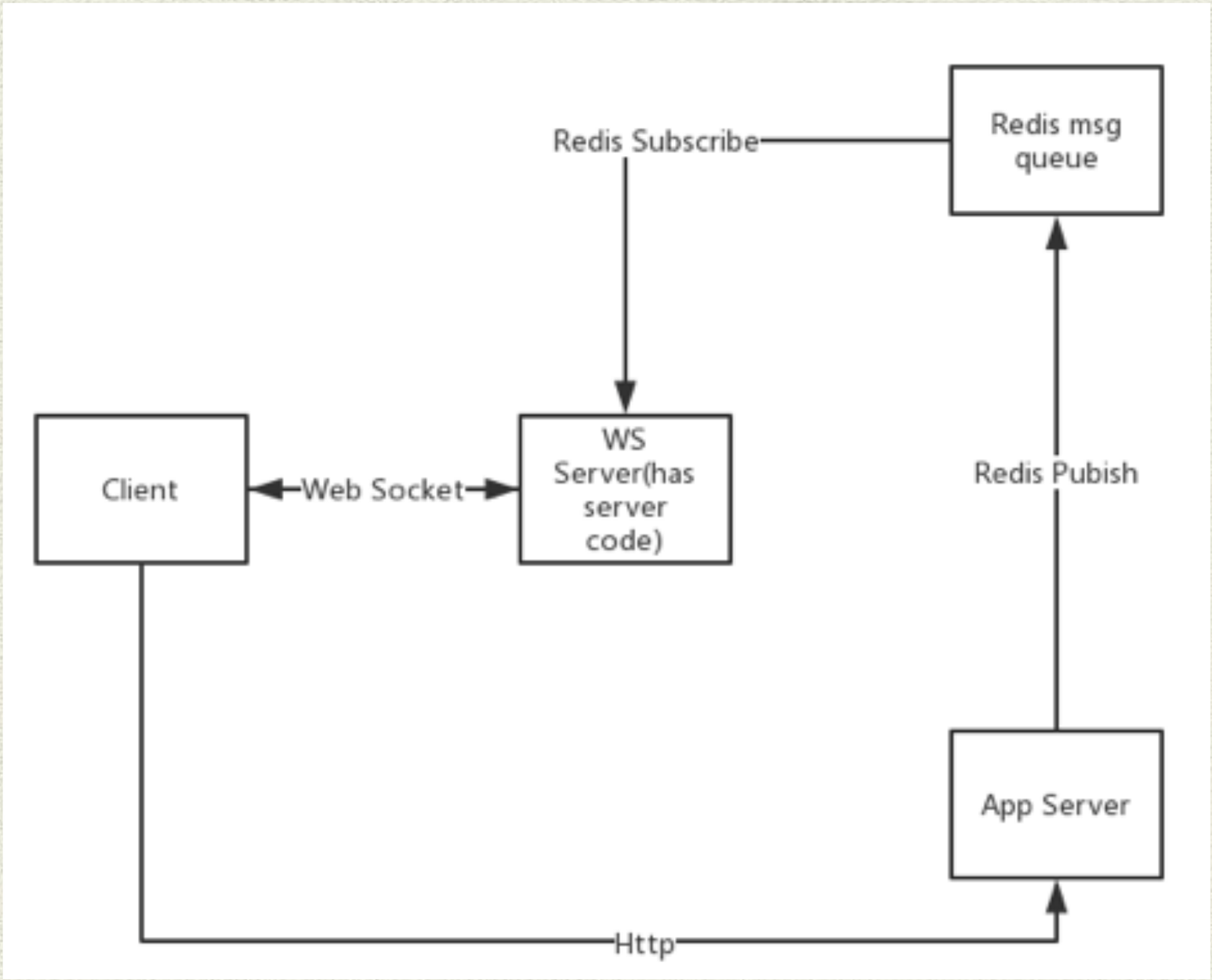
ActionCable 依赖

nio4r：处理异步 IO

websocket-driver：处理 WebSocket 协议，提供 WebSocket 事件的抽象

concurrency-ruby：处理并发，提供 thread pool、timer 、atomic、lock 等

ActionCable 架构



ActionCable Blocking I/O

Block I/O -- 线程模型

在进行 I/O 读写时，如果读写没有完成，线程会被挂起

一个线程对应一个连接

Block I/O -- 线程模型带来的问题

线程会消耗过多的内存

线程切换会造成 CPU 的浪费

最终导致无法维护大量链接

ActionCable Non-Blocking I/O

I/O 多路复用

读写方法被调用时，无论读写是否完成都直接返回

单独线程监听多个 I/O 对象

当 I/O ready 时，调用相应的 callback

用少数线程可以维护更多的连接

更少的资源维护更多的连接

ActionCable StreamEventLoop

注：为了便于理解，下面的代码均有省略

```ruby
class StreamEventLoop
  def spawn
    @nio ||= NIO::Selector.new
    @thread = Thread.new { run }
  end

  def attach(io, stream)
    @todo << lambda do
      @map[io] = @nio.register(io, :r)
      @map[io].value = stream
    end
    wakeup
  end

  def run
    loop do
      until @todo.empty?
        @todo.pop(true).call
      end

      next unless monitors = @nio.select

      monitors.each do |monitor|
        io = monitor.io
        stream = monitor.value

        if monitor.writable?
          if stream.flush_write_buffer
            monitor.interests = :r
          end
          next unless monitor.readable?
        end

        incoming = io.read_nonblock(4096, exception: false)
        case incoming
        when :wait_readable
          next
        when nil
          stream.close
        else
          stream.receive incoming
        end
      end
    end
  end
end
```

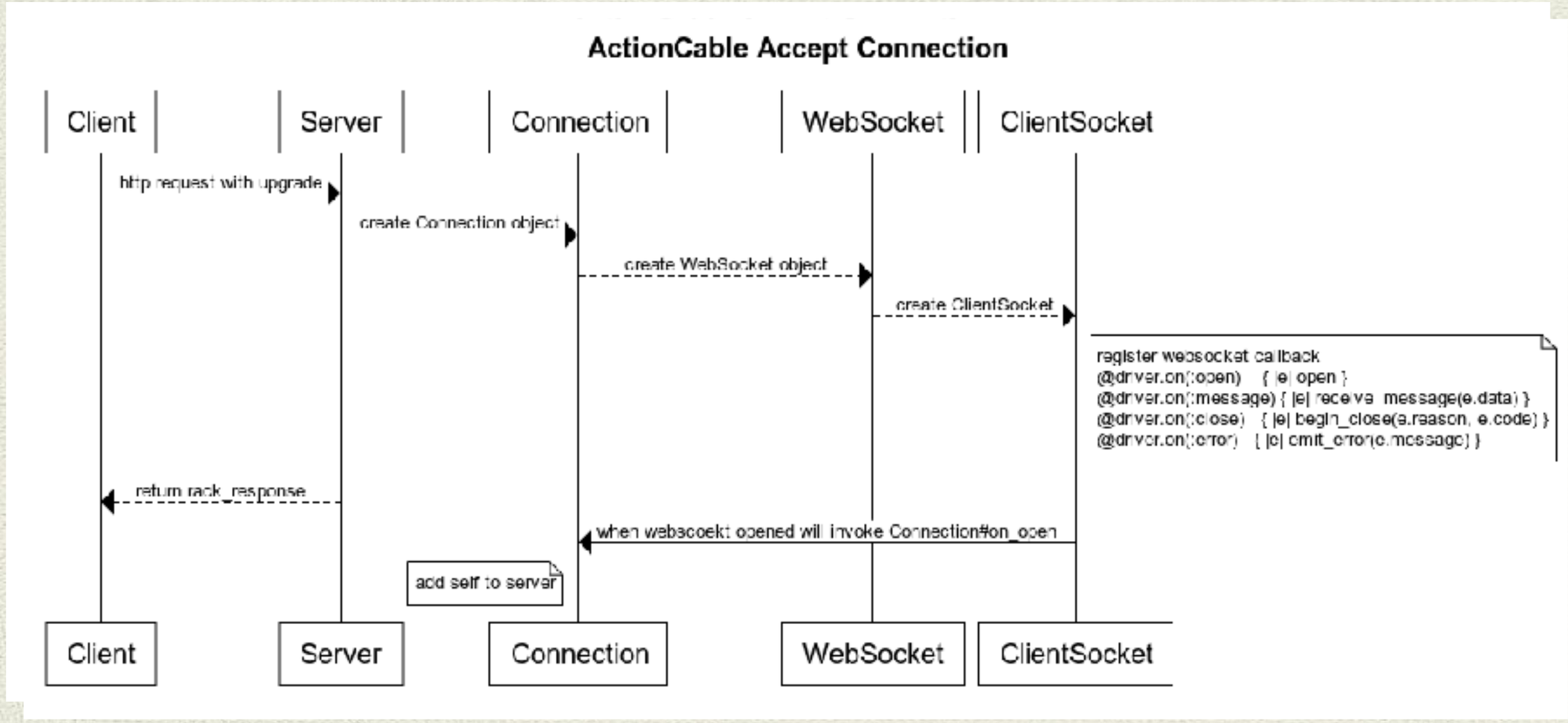
ActionCable 创建连接

接受客户端的请求

创建 Connection 对象 并被记录到 Server

将 IO 挂载到 stream_event_loop

ActionCable 创建连接



ActionCable 创建连接

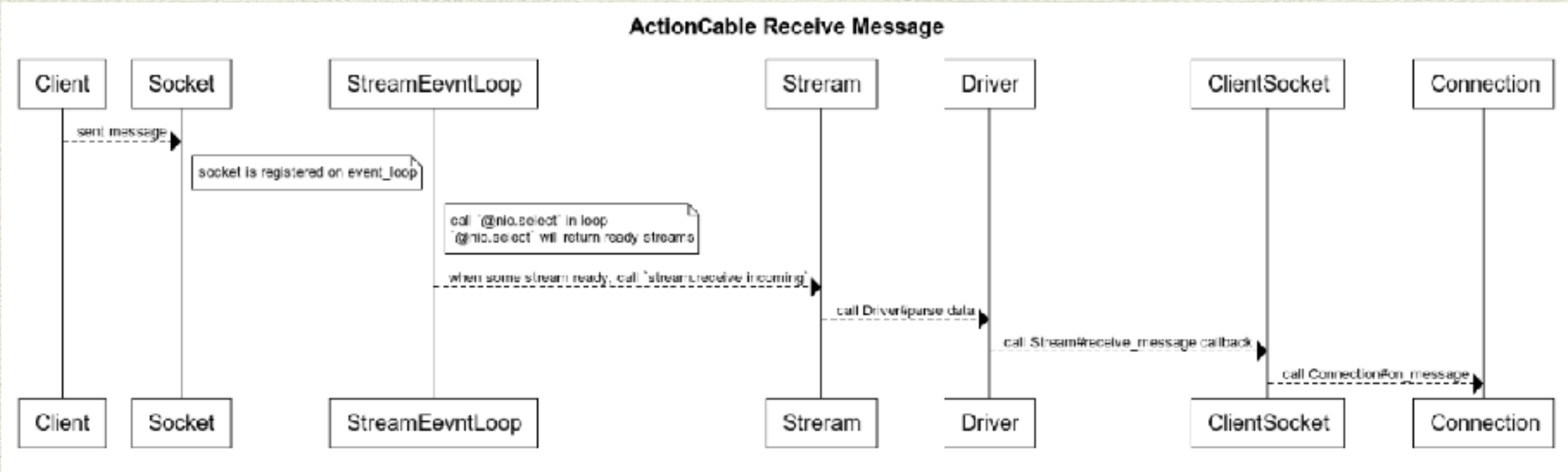
```ruby
module Server
  class Base
    def call(env)
      setup_heartbeat_timer
      config.connection_class.call.new(self, env).process
    end
  end
end

module Connection
  class Base
    def initialize(server, env, coder: ActiveSupport::JSON)
      @server, @env, @coder = server, env, coder

      @worker_pool = server.worker_pool
      @logger = new_tagged_logger

      @websocket      = ActionCable::Connection::WebSocket.new(env, self, event_loop)
      @subscriptions  = ActionCable::Connection::Subscriptions.new(self)
      @message_buffer = ActionCable::Connection::MessageBuffer.new(self)

      @_internal_subscriptions = nil
      @started_at = Time.now
    end

    def process
      respond_to_successful_request
    end

    def respond_to_successful_request
      websocket.rack_response
    end

    def on_open
      send_async :handle_open
    end

    def handle_open
      @protocol = websocket.protocol
      connect if respond_to?(:connect)
      subscribe_to_internal_channel
      server.add_connection(self)
    end
  end
end
```

ActionCable 创建连接

```
class WebSocket
  def initialize(env, event_target, event_loop, protocols: ActionCable::INTERNAL[:protocols])
    @websocket = ClientSocket.new(env, event_target, event_loop, protocols)
  end

  def rack_response
    websocket.rack_response
  end
end

class ClientSocket
  def initialize(env, event_target, event_loop, protocols)
    @env          = env
    @event_target = event_target
    @event_loop   = event_loop

    @url = ClientSocket.determine_url(@env)

    @driver = @driver_started = nil
    @close_params = ["", 1006]

    @ready_state = CONNECTING

    @driver = ::WebSocket::Driver.rack(self, protocols: protocols)

    @driver.on(:open)    { |e| open }
    @driver.on(:message) { |e| receive_message(e.data) }
    @driver.on(:close)   { |e| begin_close(e.reason, e.code) }
    @driver.on(:error)   { |e| emit_error(e.message) }

    @stream = ActionCable::Connection::Stream.new(@event_loop, self)
  end

  def start_driver
    return if @driver.nil? || @driver_started
    @stream.hijack_rack_socket

    if callback = @env["async.callback"]
      callback.call([101, {}, @stream])
    end

    @driver_started = true
    @driver.start
  end

  def rack_response
    start_driver
    [ -1, {}, [] ]
  end

  def open
    return unless @ready_state == CONNECTING
    @ready_state = OPEN

    @event_target.on_open
  end
end
```

ActionCable 收消息

EventLoop -> Stream -> Driver -> ClientSocket -> Connection



ActionCable 向论组广播消息

Subscribe Channel

创建相应的 channel 对象

将 channel 对象存储到 subscriptions 内

```
module Connection
  class Base
    def receive(websocket_message)
      send_async :dispatch_websocket_message, websocket_message
    end

    def dispatch_websocket_message(websocket_message)
      subscriptions.execute_command decode(websocket_message)
    end
  end

  class Subscriptions
    def execute_command(data)
      case data["command"]
      when "subscribe"   then add data
      when "unsubscribe" then remove data
      when "message"     then perform_action data
      end

      def add(data)
        subscription = subscription_klass.new(connection, data["identifier"], id_options)
        subscriptions[id_key] = subscription
        subscription.subscribe_to_channel
      end
    end
  end
end
```

ActionCable 向论组广播消息

Create Stream

建立 stream 到 connection 之间的关系

```ruby
module Channel
  def stream_from(broadcasting, callback = nil, coder: nil, &block)
    connection.server.event_loop.post do
      pubsub.subscribe(broadcasting, callback, success_callback)
    end
  end
end

module SubscriptionAdapter
  class Redis < Base
    def subscribe(channel, callback, success_callback = nil)
      listener.add_subscriber(channel, callback, success_callback)
    end
  end
end

class Listener < SubscriberMap
end

class SubscriberMap
  def initialize
    @subscribers = Hash.new { |h, k| h[k] = [] }
    @sync = Mutex.new
  end

  def add_subscriber(channel, subscriber, on_success)
    @sync.synchronize do
      new_channel = !@subscribers.key?(channel)

      @subscribers[channel] << subscriber

      if new_channel
        add_channel channel, on_success
      elsif on_success
        on_success.call
      end
    end
  end
end
```

ActionCable 向论组广播消息

Broadcast Message

```ruby
module Server
  class Broadcaster
    attr_reader :server, :broadcasting, :coder

    def initialize(server, broadcasting, coder:)
      @server, @broadcasting, @coder = server, broadcasting, coder
    end

    def broadcast(message)
      server.pubsub.broadcast broadcasting, encoded
    end
  end
end

module ActionCable
  module SubscriptionAdapter
    class Redis < Base
      def broadcast(channel, payload)
        redis_connection_for_broadcasts.publish(channel, payload)
      end
    end
  end
end
```

ActionCable 向论组广播消息

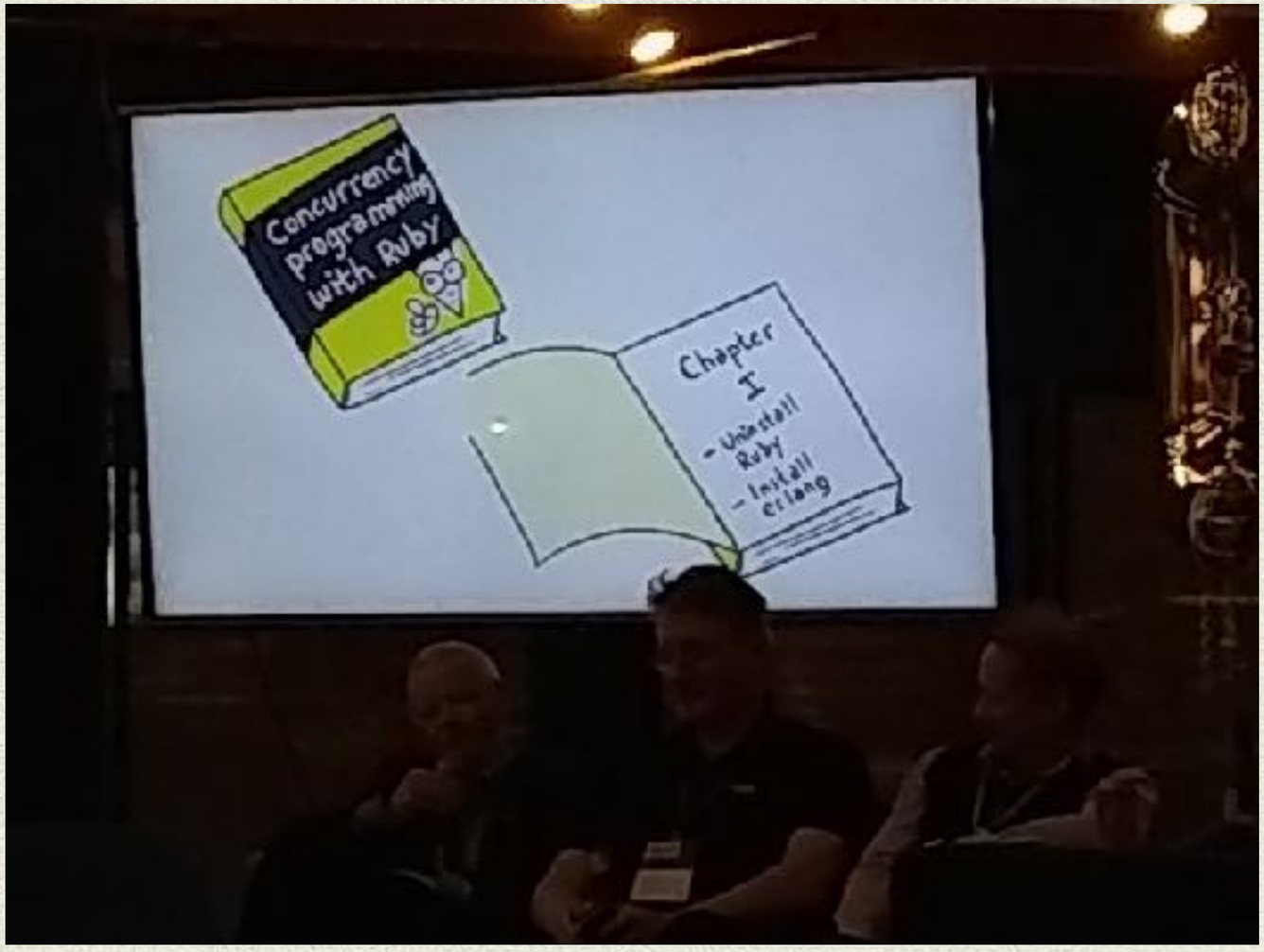
Consume Message

```ruby
module SubscriptionAdapter
  class Redis < Base
    class Listener < SubscriberMap
      def listen(conn)
        on.message do |chan, message|
          broadcast(chan, message)
        end
      end
    end
  end

  class SubscriberMap
    def broadcast(channel, message)
      list = @sync.synchronize do
        return if !@subscribers.key?(channel)
        @subscribers[channel].dup
      end

      list.each do |subscriber|
        invoke_callback(subscriber, message)
      end
    end
  end
end
```

AnyCable

Why AnyCable?

Ruby 并不擅长处理并发



Why AnyCable?

Ruby 并不擅长处理并发

Erlang：说的好像其他语言把并发处理的很好一样

如果把语言比作汽车的话，好的赛车同时也需要好的车手

并发对任何语言都不是一件容易的事情

Why AnyCable?

AnyCable 可以利用其他语言的优势

Why Rust?

性能好

静态语言

没有 GC

安全

严格的编译器

Rust 简介

变量所有权（variable ownership）

每个变量都有一个所有者

一个变量只能有一个所有者

如果所有者，超出了其定义的范围，则会被收回

Rust 简介

变量所有权（variable ownership）

```
let s1 = String::from("hello");
let s2 = s1;

println!("{}, world!", s1);

error[E0382]: use of moved value: `s1`
--> src/main.rs:5:28
  |
3 |     let s2 = s1;
  |         -- value moved here
4 |
5 |     println!("{}, world!", s1);
  |                            ^^ value used here after move
  |
  = note: move occurs because `s1` has type `std::string::String`, which does
  not implement the `Copy` trait
```

Rust 简介

代码更明确

变量默认不可变，可变变量需要用 mut 声明

以 hash 为例，当获取某个 key 时

需要考虑这个 key 存在和不存在的情况

Rust 简介

代码更明确

```rust
let map = HashMap::new();
map.insert(1, "a");

// let map = HashMap::new();
//     --- help: consider changing this to be mutable: `mut map`
// map.insert(1, "a");
// ^^^ cannot borrow as mutable

let mut map = HashMap::new();
map.insert(1, "a");

match map.get(&1) {
    Some(v) =>
        println!("fond: {}", v),
    None =>
        println!("not found"),
}
```

Rust 简介

安全

编译器会帮助我们检查问题

```rust
let map = Arc::new(Mutex::new(HashMap::new()));
map.insert(1, "a");

// error[E0599]: no method named `insert` found for type
// `std::sync::Arc<std::sync::Mutex<std::collections::HashMap<_, _>>>`
// in the current scope
// map.insert(1, "a");
//     ^^^^^^

let map = Arc::new(Mutex::new(HashMap::new()));
map.lock().unwrap().insert(1, "a");
```

Rust 简介

静态类型 & 类型推断

变量需要声明类型

编译器会尽可能帮我们做推断

```
let map = HashMap::new();

// let map = HashMap::new();
//     ---   ^^^^^^^^^^^^^ cannot infer type for `K`
//     consider giving `map` the explicit type `std::collections::HashMap<K, V>`,
//     where the type parameter `K` is specified

let mut map = HashMap::new();
map.insert(1, "a");
```

Tokio 简介

Rust 异步编程运行时

一个物理线程对应多个 tokio task

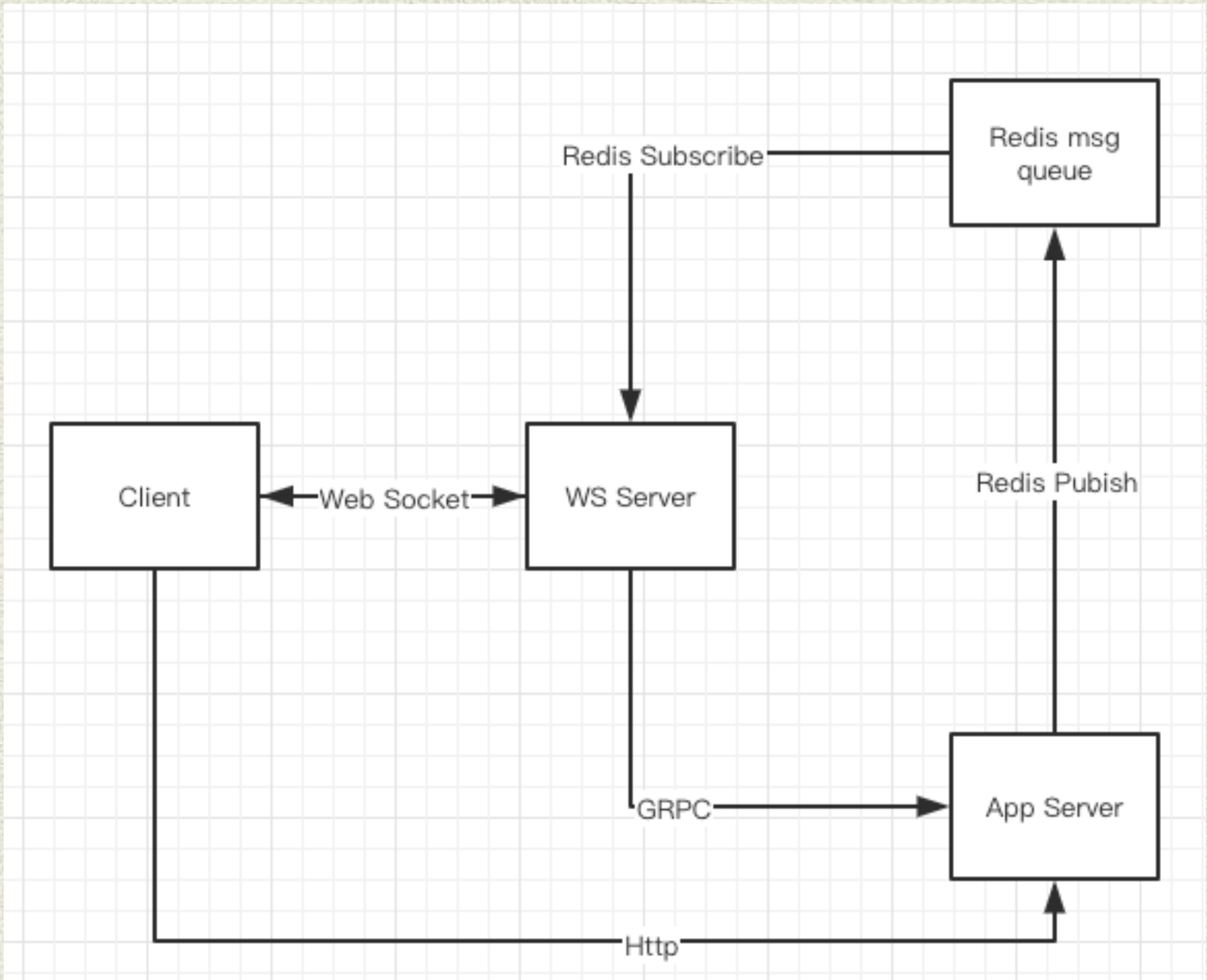
提供了简单的并发模型

抽象了异步 I/O

一个简单的例子

```
let ping_interval = Interval::new_interval(Duration::from_millis(PING_INTERVAL_MS))
    .for_each(move |_| {
        // send ping to every connection
        Ok(())
    }).map_err(|_e| ());

tokio::spawn(ping_interval);
```

AnyCable 架构


Redis Subscribe
Redis msg queue
Client
Web Socket
WS Server
Redis Pubish
GRPC
App Server
Http

AnyCable Rust

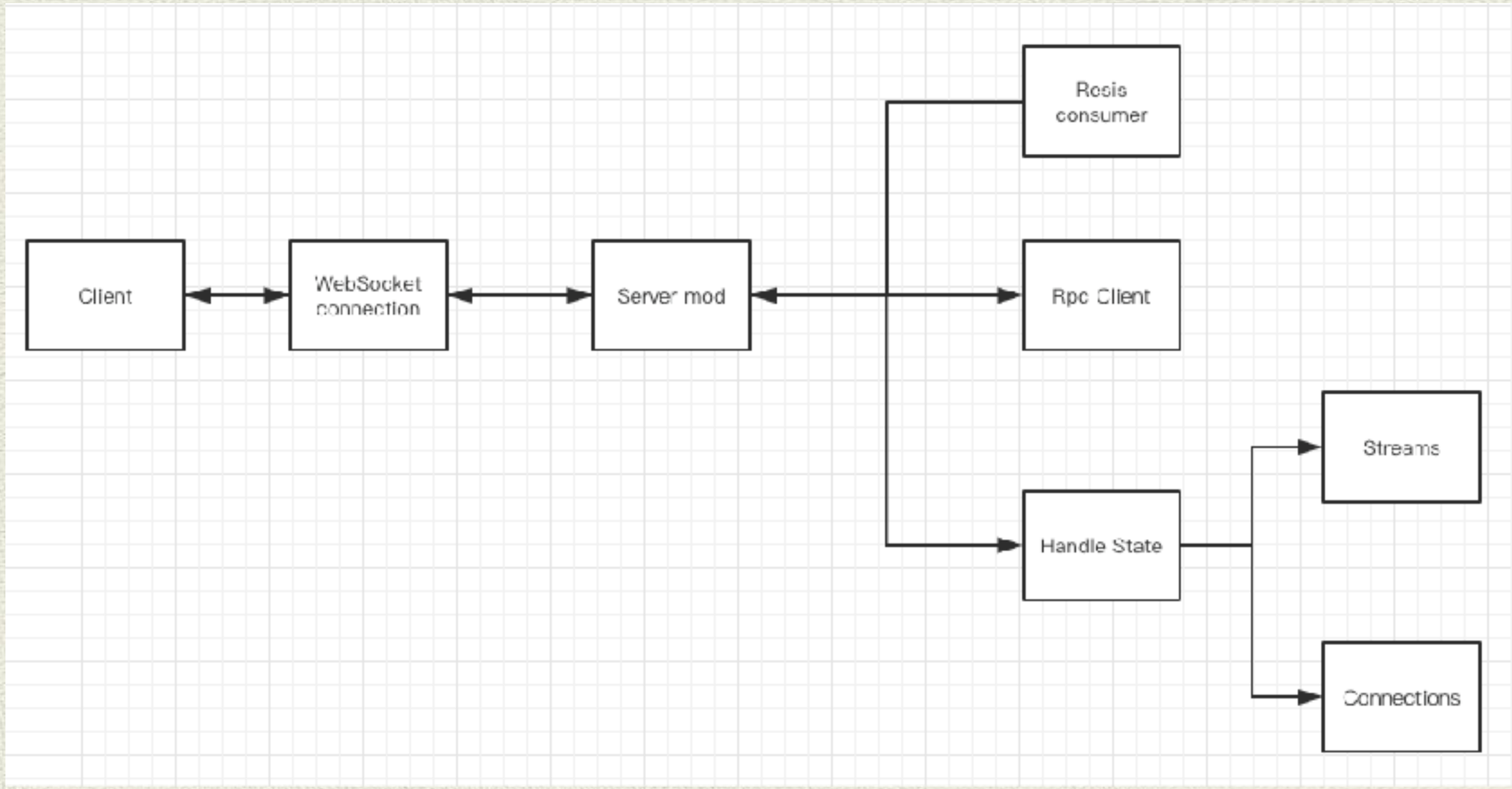
WIP

需要压测和优化

主要是用来学习 Rust

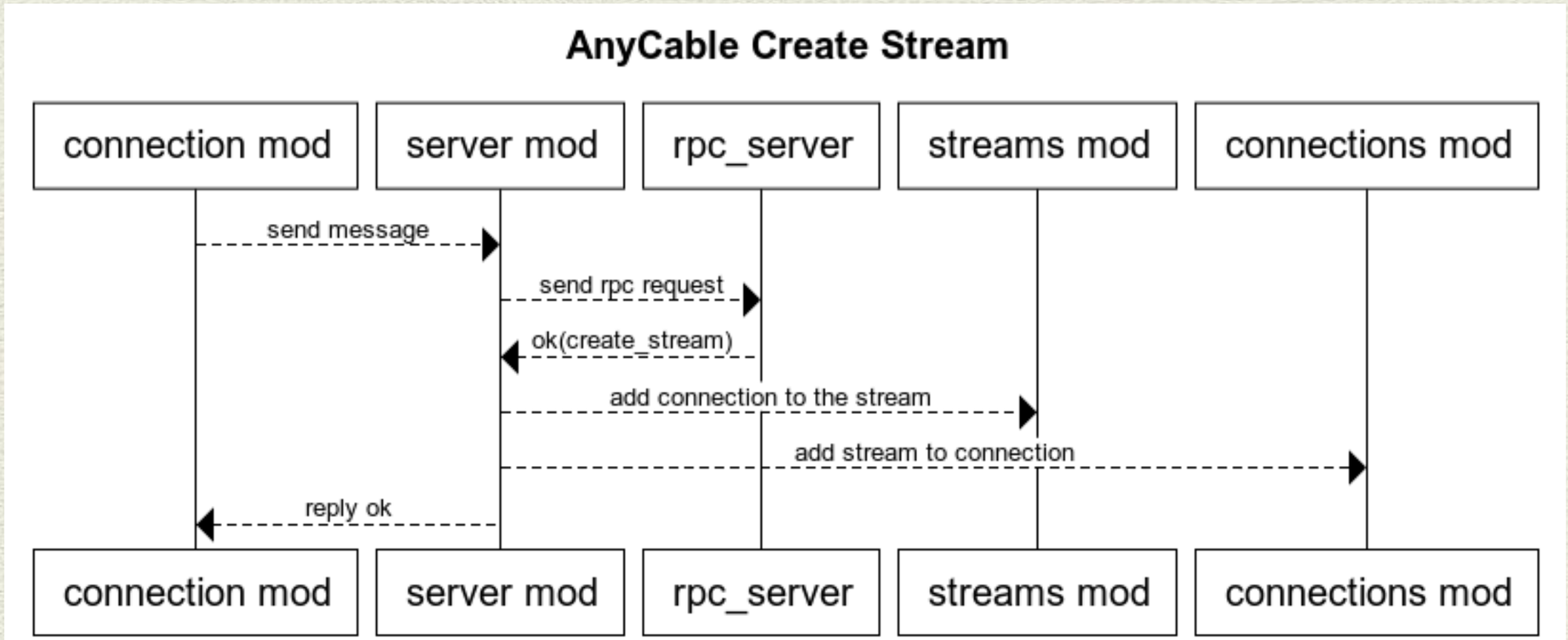
AnyCable Rust 简介

代码结构



AnyCable Rust 简介

创建 stream



AnyCable Rust 简介

创建 stream

```
let ws_reader = stream.for_each(move |message: Message| {
    let data = message.to_text().unwrap();

    server.lock().unwrap().
        receive_message(addr, data);

    Ok(())
});
```

AnyCable Rust 简介

创建 stream

```
pub fn receive_message(&mut self, addr: SocketAddr, message: &str) {
    let v: Value = serde_json::from_str(message).unwrap();
    let command = &v["command"];
    let mut command_data = "";

    if command == "message" {
        command_data = &v["data"].as_str().unwrap();
    }

    let channel = &v["identifier"];
    let identifiers = self.connections.get_conn_identifiers(&addr);

    let reply = self.rpc_client.
        command(command.as_str().unwrap().to_string(),
                identifiers.to_string(),
                channel.as_str().unwrap().to_string(),
                command_data.to_string());

    self.handle_rpc_command_resp(addr,
                                 channel.as_str().unwrap().to_string(),
                                 reply);
}
```

AnyCable Rust 简介

创建 stream

```
fn start_channel_streams(
    &mut self,
    addr: SocketAddr,
    channel: String,
    respone: CommandResponse) {

    for stream in respone.get_streams().iter() {
        self.streams.put_stream(stream, addr, channel.to_string());

        self.connections.add_stream_to_conn(&addr, stream.to_string(), channel.to_string());
    }
}
```

相关连接

AnyCable

anycable-rs

支撑ActionCable的底层库

HTTP Request Demo by Future and Nio4r

Fearless Concurrency with Rust

Thanks