



# Надежно пишем в сокет

Как Rust помогает писать  
многопоточный асинхронный код

Updated in March 2025



**Денис Давыдов**  
Systems Engineer,  
Cloudflare, FL Team  
UK, Лондон



@andrushaTheSlayer



davidovdd92@mail.ru



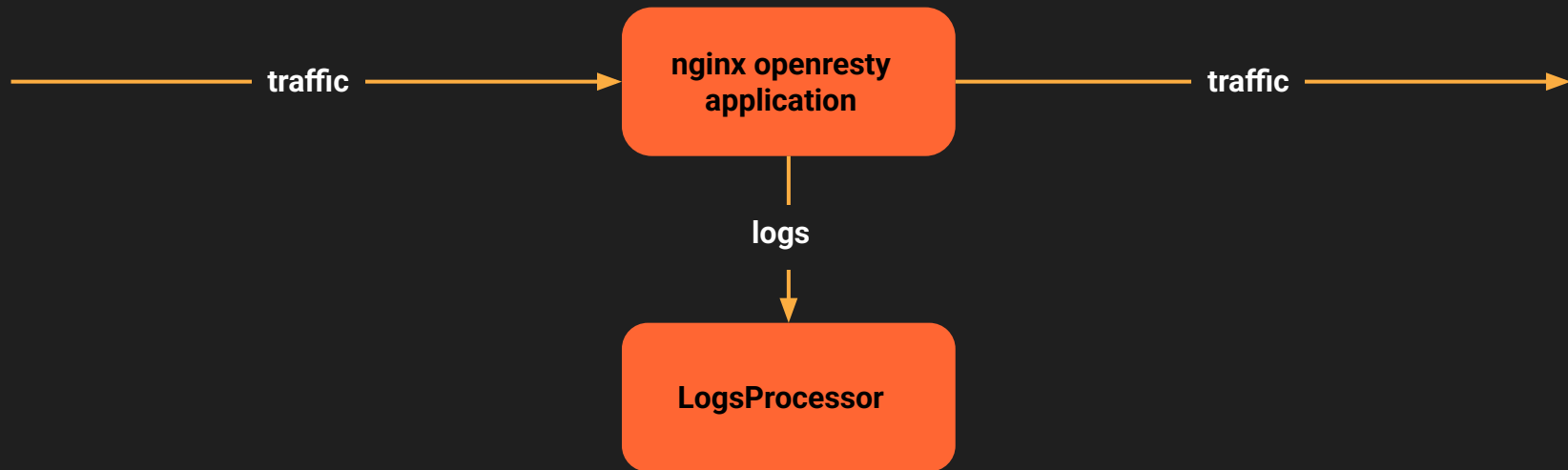
<https://github.com/WowVeryLogin>

# Agenda

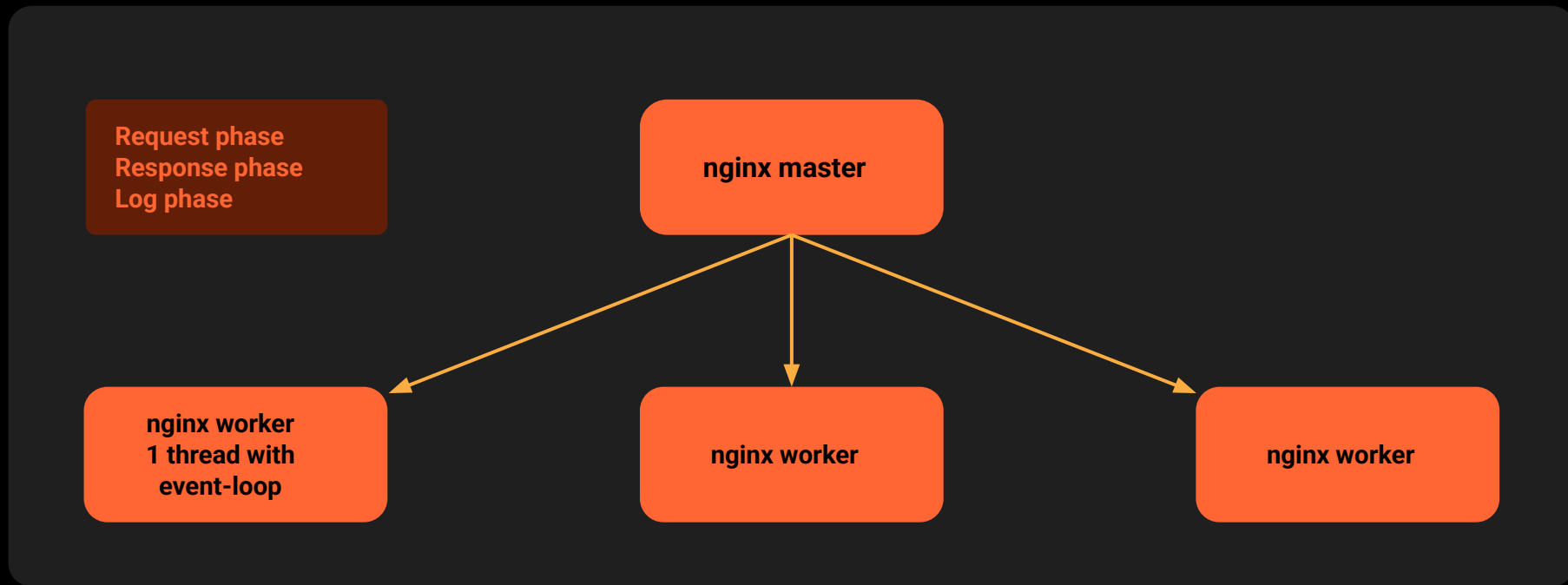
- 1 Постановка проблемы
- 2 Немного об асинхронности
- 3 Что внутри у Tokio
- 4 Первое решение
- 5 Синхронизация
- 6 Упрощаем решение
- 7 Про тестирование

# Проблема

# Ngіnх и логи

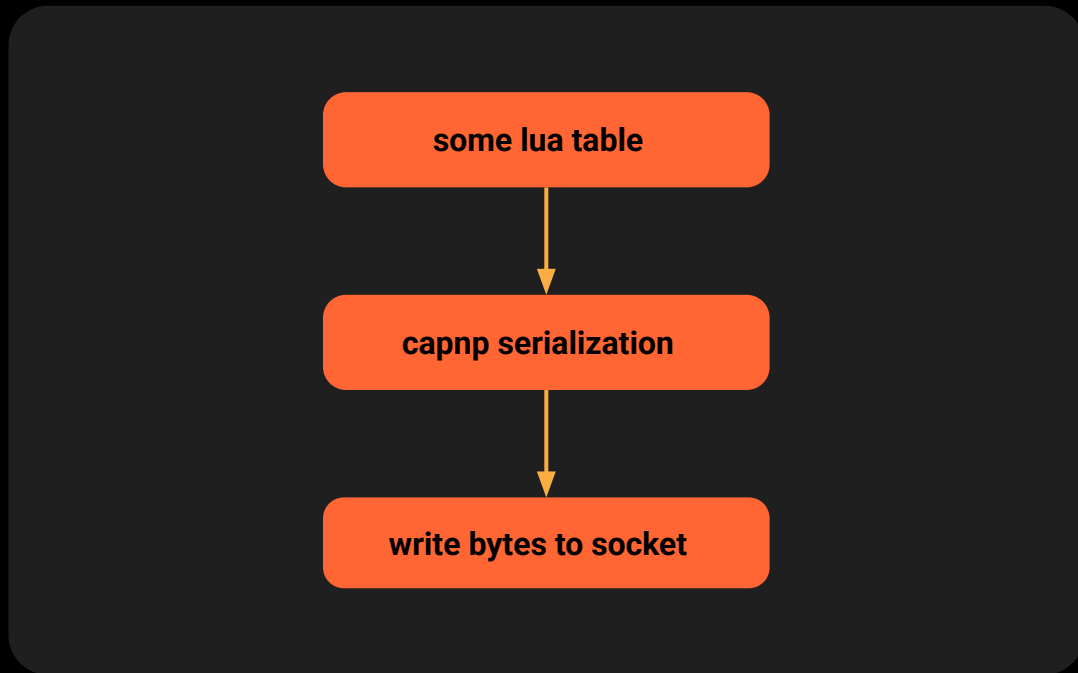


# Ngіnx и логи



# Nginx

- Нет IO в log-phase
- Пока мы делаем что-то, мы не сидим в event-loopе и не принимаем коннекты
- Вопрос миграции



# Начинаем дизайнИТЬ

**Nginx worker**

**Log Data**

**Magical rust library**

# Rust легко интегрировать

Lua code

FFI Layer  
Bindgen and  
luajit cAPI

Rust World

# Bindgen

## The Virgin Python

Has a bunch of weird syntax and data types, doesn't know how to use most of them

Uses "import" instead of writing actual code

Close to being the slowest scripting language

Thinks that Python can do everything

Wastes time figuring where scopes end

Interpreter is at least a megabyte

Embedding an interpreter takes hours of set up



## The Chad Lua

Has only 8 types

Started career as a Roblox hacker

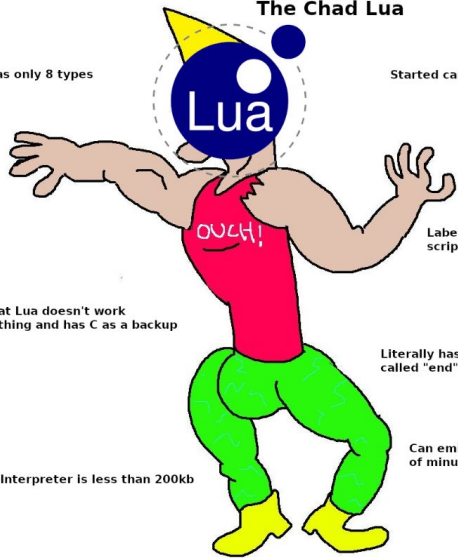
Labelled as the fastest scripting language

Knows that Lua doesn't work for everything and has C as a backup

Literally has a keyword called "end"

Interpreter is less than 200kb

Can embed an interpreter in a matter of minutes



# Bindgen

④ build.rs > ...

You, 14 months ago | 1 author (You)

```
1 extern crate bindgen;
2
3 use std::env;
4 use std::path::PathBuf;
5
6 fn link_lua_api() {
7     println!("cargo:rerun-if-changed=wrapper.h");
8
9     let bindings = Bindings = bindgen::Builder::default() Builder
10     .header("wrapper.h") Builder
11     .parse_callbacks(cb: Box::new(bindgen::CargoCallbacks::new())) Builder
12     .generate() Result<Bindings, BindgenError>
13     .expect(msg: "Unable to generate bindings");
14
15     let out_path: PathBuf = PathBuf::from(env::var(key: "OUT_DIR").unwrap());
16     bindings.write_to_file(path: out_path.join(path: "bindings.rs")) Result<, Error>
17     .expect(msg: "Couldn't write bindings!");
18 }
19
20
21 fn main() {
22     println!("cargo:rerun-if-changed=build.rs");
23     link_lua_api();
24 }
```

```
2658 extern "C" {
2659     pub fn lua_tonumber(L: *mut lua_State, idx: ::std::os::raw::c_int) -> lua_Number;
2660 }
2661 extern "C" {
2662     pub fn lua_tointeger(L: *mut lua_State, idx: ::std::os::raw::c_int) -> lua_Integer;
2663 }
2664 extern "C" {
2665     pub fn lua_toboolean(L: *mut lua_State, idx: ::std::os::raw::c_int) -> ::std::os::raw::c_int;
2666 }
2667 extern "C" {
2668     pub fn lua_tolstring(
2669         L: *mut lua_State,
2670         idx: ::std::os::raw::c_int,
2671         len: *mut usize,
2672     ) -> *const ::std::os::raw::c_char;
2673 }
2674 extern "C" {
2675     pub fn lua_objlen(L: *mut lua_State, idx: ::std::os::raw::c_int) -> usize;
2676 }
2677 extern "C" {
2678     pub fn lua_tocfunction(L: *mut lua_State, idx: ::std::os::raw::c_int) -> lua_CFunction;
2679 }
2680 extern "C" {
2681     pub fn lua_touserdata(
2682         L: *mut lua_State,
2683         idx: ::std::os::raw::c_int,
2684     ) -> *mut ::std::os::raw::c_void;
2685 }
2686 extern "C" {
2687     pub fn lua_tothread(L: *mut lua_State, idx: ::std::os::raw::c_int) -> *mut lua_State;
2688 }
```

# Bindgen

```
60  /// # Safety
61  ///
62  /// Provide valid pointer to socket pool object and message string with length
63  ///
64  #[allow(non_snake_case)]
65  #[no_mangle]
66  pub unsafe extern "C" fn lua_log_cdata(L: *mut lua_State) -> i32 {
67  --- let msg_len: isize = lua_tointeger(L, idx: -1);
68  --- let msg: *const i8 = lua_tolstring(L, idx: -2, len: std::ptr::null_mut());
69  --- let log_socket_id: isize = lua_tointeger(L, idx: -3);
70  --- let pool: &mut Pool = (lua_touserdata(L, idx: -4) as *mut Pool).as_mut().unwrap();
71
72  --- let data: &[u8] = std::slice::from_raw_parts(data: msg as *mut u8, msg_len as usize);
73  --- let bytes_sent: Result<usize, String> = pool.log_cdata(log_socket_id as usize, msg: data);
74
75  --- lua_settop(L, idx: -5); // remove pool, msg, msg_len from lua stack
76  --- match bytes_sent {
77  ---     Ok(bytes_sent: usize) => {
78  ---         --- lua_pushinteger(L, n: bytes_sent as isize);
79  ---         --- lua_pushnil(L);
80  ---     }
81  ---     Err(err: String) => {
82  ---         --- lua_pushinteger(L, n: 0isize);
83  ---         --- lua_pushlstring(L, s: err.as_ptr().cast(), l: err.len());
84  ---     }
85  --- };
86  --- 2
87  }
```

# Немного об асинхронности

# Что у нас есть

1 trait Future

2 async/await

# Future

```
pub enum Poll<T> {  
    Ready(T),  
    Pending,  
}
```

```
pub trait Future {  
    type Output;  
  
    // Required method  
    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Output>;  
}
```

# Async/await

```
3 45  ✓ async fn my_func() {  
4 46     ... // some calculations  
5 47     ... inner_func().await;  
6 48     ... // some more calculations  
7 49     ... another_inner_func().await;  
8 50 }
```

# Async/await

```
3 45 | async fn my_func() {  
4 46 |     ...// some calculations  
5 47 |     ...inner_func().await;  
6 48 |     ...// some more calculations  
7 49 |     ...another_inner_func().await;  
8 50 | }
```

```
2 52 | // Different states of the future  
    0 implementations  
3 53 | enum State {  
4 54 |     ... Start,  
5 55 |     ... WaitingForFuture1,  
6 56 |     ... WaitingForFuture2,  
7 57 |     ... Done,  
8 58 | }  
9 59 |  
    You, 1 second ago | 1 author (You) | 1 implementation  
10 60 | struct MyFuncFuture {  
11 61 |     ... state: State,  
12 62 |     ... future1: Option<InnerFuncFuture>,  
13 63 |     ... future2: Option<AnotherInnerFuncFuture>,  
14 64 | }  
15 65 |  
16 66 | impl Future for MyFuncFuture {  
17 67 |     ... type Output = ();  
18 68 |  
19 69 |     ... fn poll(mut self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Output> {
```

# Async/await

```

45 | ✓ async fn my_func() {
46 |     ... // some calculations
47 |     ... inner_func().await;
48 |     ... // some more calculations
49 |     ... another_inner_func().await;
50 | }
    
```

```

66 | impl Future for MyFuncFuture {
67 |     type Output = ();
68 |
69 |     fn poll(mut self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Output> {
70 |         loop {
71 |             match self.state {
72 |                 State::Start => {
73 |                     // some calculations...
74 |                     self.future1 = Some(inner_func()); // Store future
75 |                     self.state = State::WaitingForFuture1;
76 |                 }
77 |                 State::WaitingForFuture1 => {
78 |                     // Poll `future1`
79 |                     if let Some(future1: &mut InnerFuncFuture) = self.future1.as_mut() {
80 |                         let pinned_future1: Pin<&mut InnerFuncFuture> = unsafe { Pin::new_unchecked(pointer: future1) };
81 |                         if let Poll::Ready(()) = pinned_future1.poll(cx) {
82 |                             self.future1 = None;
83 |                             self.state = State::WaitingForFuture2;
84 |                         } else {
85 |                             return Poll::Pending; // Not ready yet
86 |                         }
87 |                     }
88 |                 }
89 |                 State::WaitingForFuture2 => {
90 |                     // some more calculations...
91 |                     let future2: AnotherInnerFuncFuture = another_inner_func(); // Call second async function
92 |                     self.future2 = Some(future2);
93 |                     self.state = State::Done;
94 |                 }
95 |                 State::Done => {
96 |                     if let Some(future2: &mut AnotherInnerFuncFuture) = self.future2.as_mut() {
97 |                         let pinned_future2: Pin<&mut AnotherInnerFuncFuture> = unsafe { Pin::new_unchecked(pointer: future2) };
98 |                         if let Poll::Ready(()) = pinned_future2.poll(cx) {
99 |                             return Poll::Ready(()); // Finished execution
100 |                        } else {
101 |                            return Poll::Pending;
102 |                        }
103 |                    }
104 |                }
105 |            }
106 |        }
107 |     } fn poll
108 | } impl Future for MyFuncFuture
    
```

# Async/await

```

45 | ✓ async fn my_func() {
46 |     // some calculations
47 |     inner_func().await;
48 |     // some more calculations
49 |     another_inner_func().await;
50 | }

```

```

66 | impl Future for MyFuncFuture {
67 |     type Output = ();
68 |
69 |     fn poll(mut self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Output> {
70 |         loop {
71 |             match self.state {
72 |                 State::Start => {
73 |                     // some calculations...
74 |                     self.future1 = Some(inner_func()); // Store future
75 |                     self.state = State::WaitingForFuture1;
76 |                 }
77 |                 State::WaitingForFuture1 => {
78 |                     // Poll `future1`
79 |                     if let Some(future1: &mut InnerFuncFuture) = self.future1.as_mut() {
80 |                         let pinned_future1: Pin<&mut InnerFuncFuture> = unsafe { Pin::new_unchecked(pointer: future1) };
81 |                         if let Poll::Ready(()) = pinned_future1.poll(cx) {
82 |                             self.future1 = None;
83 |                             self.state = State::WaitingForFuture2;
84 |                         } else {
85 |                             return Poll::Pending; // Not ready yet
86 |                         }
87 |                     }
88 |                 }
89 |                 State::WaitingForFuture2 => {
90 |                     // some more calculations...
91 |                     let future2: AnotherInnerFuncFuture = another_inner_func(); // Call second async function
92 |                     self.future2 = Some(future2);
93 |                     self.state = State::Done;
94 |                 }
95 |                 State::Done => {
96 |                     if let Some(future2: &mut AnotherInnerFuncFuture) = self.future2.as_mut() {
97 |                         let pinned_future2: Pin<&mut AnotherInnerFuncFuture> = unsafe { Pin::new_unchecked(pointer: future2) };
98 |                         if let Poll::Ready(()) = pinned_future2.poll(cx) {
99 |                             return Poll::Ready(()); // Finished execution
100 |                        } else {
101 |                            return Poll::Pending;
102 |                        }
103 |                    }
104 |                }
105 |            }
106 |        }
107 |     } fn poll
108 | } impl Future for MyFuncFuture

```

# Async/await

```

45 | ✓ async fn my_func() {
46 |     ... // some calculations
47 |     ... inner_func().await;
48 |     ... // some more calculations
49 |     ... another_inner_func().await;
50 | }

```

```

66 | impl Future for MyFuncFuture {
67 |     type Output = ();
68 |
69 |     fn poll(mut self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Output> {
70 |         loop {
71 |             match self.state {
72 |                 State::Start => {
73 |                     // some calculations...
74 |                     self.future1 = Some(inner_func()); // Store future
75 |                     self.state = State::WaitingForFuture1;
76 |                 }
77 |                 State::WaitingForFuture1 => {
78 |                     // Poll `future1`
79 |                     if let Some(future1: &mut InnerFuncFuture) = self.future1.as_mut() {
80 |                         let pinned_future1: Pin<&mut InnerFuncFuture> = unsafe { Pin::new_unchecked(pointer: future1) };
81 |                         if let Poll::Ready(()) = pinned_future1.poll(cx) {
82 |                             self.future1 = None;
83 |                             self.state = State::WaitingForFuture2;
84 |                         } else {
85 |                             return Poll::Pending; // Not ready yet
86 |                         }
87 |                     }
88 |                 }
89 |                 State::WaitingForFuture2 => {
90 |                     // some more calculations...
91 |                     let future2: AnotherInnerFuncFuture = another_inner_func(); // Call second async function
92 |                     self.future2 = Some(future2);
93 |                     self.state = State::Done;
94 |                 }
95 |                 State::Done => {
96 |                     if let Some(future2: &mut AnotherInnerFuncFuture) = self.future2.as_mut() {
97 |                         let pinned_future2: Pin<&mut AnotherInnerFuncFuture> = unsafe { Pin::new_unchecked(pointer: future2) };
98 |                         if let Poll::Ready(()) = pinned_future2.poll(cx) {
99 |                             return Poll::Ready(()); // Finished execution
100 |                         } else {
101 |                             return Poll::Pending;
102 |                         }
103 |                     }
104 |                 }
105 |             }
106 |         }
107 |     } fn poll
108 | } impl Future for MyFuncFuture

```

# Async/await

```

45 | ✓ async fn my_func() {
46 |     ... // some calculations
47 |     ... inner_func().await;
48 |     ... // some more calculations
49 |     ... another_inner_func().await;
50 | }

```

```

66 | impl Future for MyFuncFuture {
67 |     type Output = ();
68 |
69 |     fn poll(mut self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Output> {
70 |         loop {
71 |             match self.state {
72 |                 State::Start => {
73 |                     // some calculations...
74 |                     self.future1 = Some(inner_func()); // Store future
75 |                     self.state = State::WaitingForFuture1;
76 |                 }
77 |                 State::WaitingForFuture1 => {
78 |                     // Poll `future1`
79 |                     if let Some(future1: &mut InnerFuncFuture) = self.future1.as_mut() {
80 |                         let pinned_future1: Pin<&mut InnerFuncFuture> = unsafe { Pin::new_unchecked(pointer: future1) };
81 |                         if let Poll::Ready(()) = pinned_future1.poll(cx) {
82 |                             self.future1 = None;
83 |                             self.state = State::WaitingForFuture2;
84 |                         } else {
85 |                             return Poll::Pending; // Not ready yet
86 |                         }
87 |                     }
88 |                 }
89 |                 State::WaitingForFuture2 => {
90 |                     // some more calculations...
91 |                     let future2: AnotherInnerFuncFuture = another_inner_func(); // Call second async function
92 |                     self.future2 = Some(future2);
93 |                     self.state = State::Done;
94 |                 }
95 |                 State::Done => {
96 |                     if let Some(future2: &mut AnotherInnerFuncFuture) = self.future2.as_mut() {
97 |                         let pinned_future2: Pin<&mut AnotherInnerFuncFuture> = unsafe { Pin::new_unchecked(pointer: future2) };
98 |                         if let Poll::Ready(()) = pinned_future2.poll(cx) {
99 |                             return Poll::Ready(()); // Finished execution
100 |                         } else {
101 |                             return Poll::Pending;
102 |                         }
103 |                     }
104 |                 }
105 |             }
106 |         }
107 |     } fn poll
108 | } impl Future for MyFuncFuture

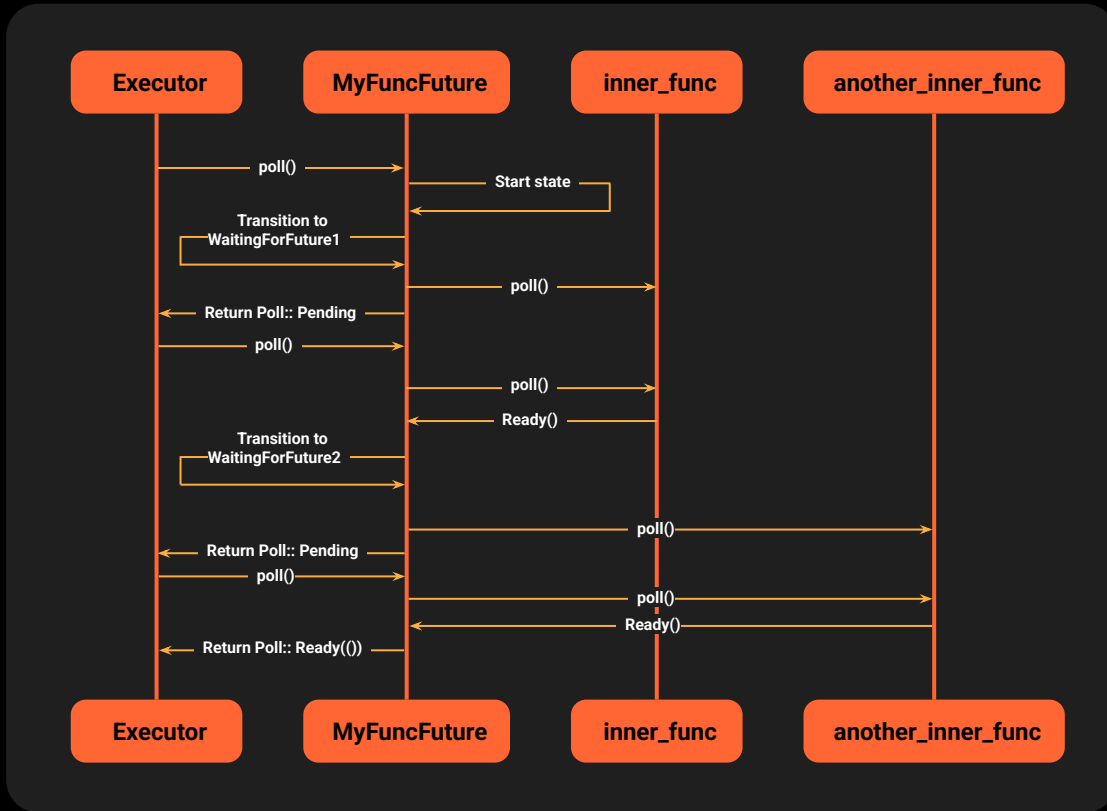
```

# Async/await

```

45  ✓ async fn my_func() {
46    ... // some calculations
47    ... inner_func().await;
48    ... // some more calculations
49    ... another_inner_func().await;
50  }

```



# Async/await

```
45 | ✓ async fn my_func() {  
46 |     ... // some calculations  
47 |     ... inner_func().await;  
48 |     ... // some more calculations  
49 |     ... another_inner_func().await;  
50 | }
```

```
2 52 | // Different states of the future  
   | 0 implementations  
3 53 | enum State {  
4 54 |     ... Start,  
5 55 |     ... WaitingForFuture1,  
6 56 |     ... WaitingForFuture2,  
7 57 |     ... Done,  
8 58 | }  
9 59 |  
   | You, 1 second ago | 1 author (You) | 1 implementation  
10 60 | struct MyFuncFuture {  
11 61 |     ... state: State,  
12 62 |     ... future1: Option<InnerFuncFuture>,  
13 63 |     ... future2: Option<AnotherInnerFuncFuture>,  
14 64 | }  
15 65 |  
16 66 | impl Future for MyFuncFuture {  
17 67 |     ... type Output = ();  
18 68 |  
19 69 |     ... fn poll(mut self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Output> {
```

```
12117 | // Function to return the desugared future  
11118 | ✓ fn my_func() -> MyFuncFuture {  
10119 |     ✓ MyFuncFuture {  
9 120 |         ... state: State::Start,  
8 121 |         ... future1: None,  
7 122 |         ... future2: None,  
6 123 |     }  
5 124 | }
```

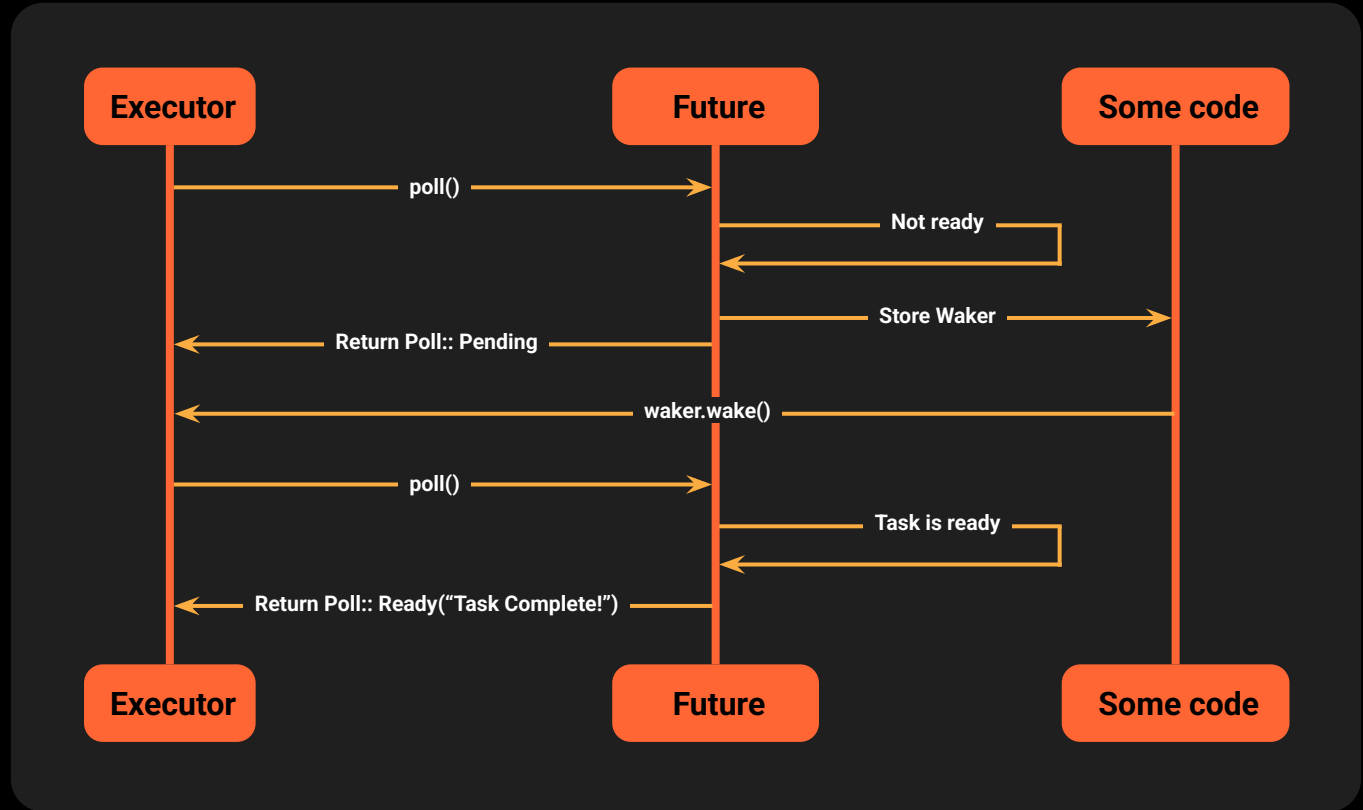
# Waker

**Future**

**waker()**

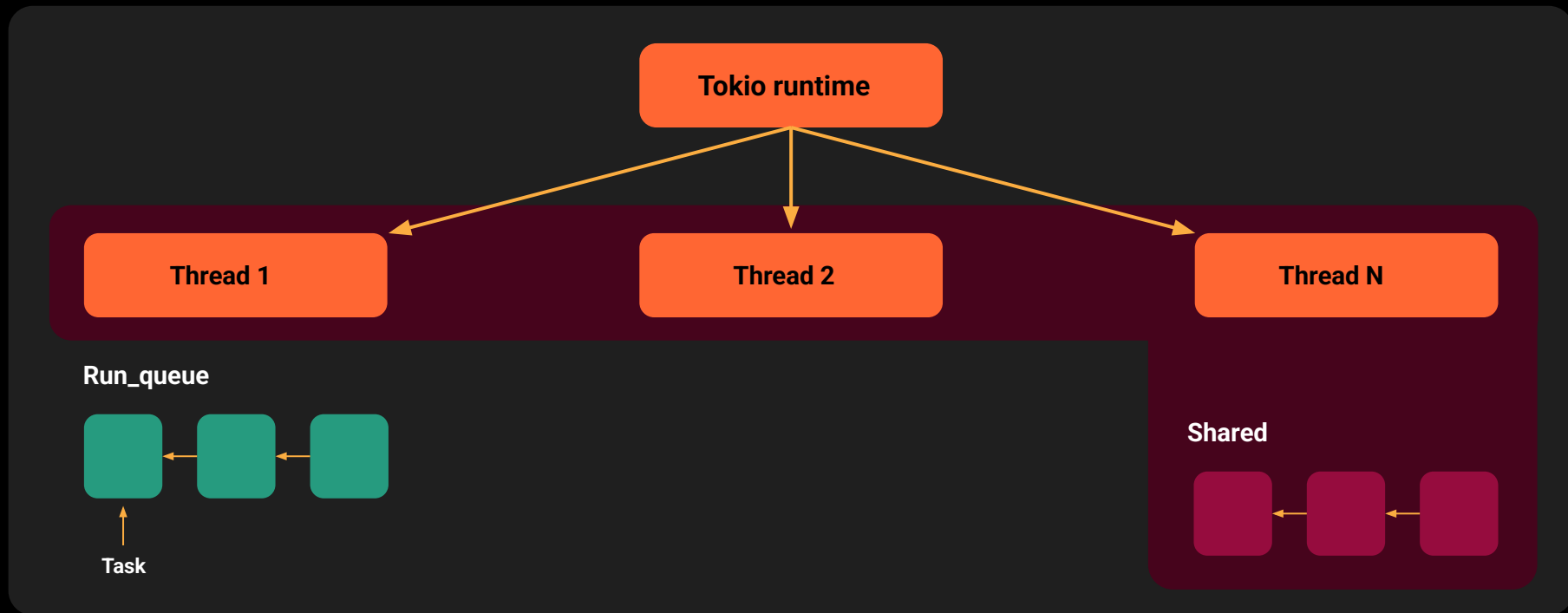
**Async Runtime**

# Waker



# Tokio

# Tokio



# Tokio

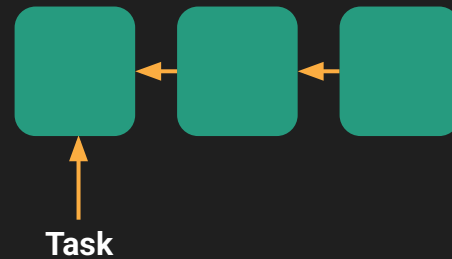
Pending Tasks



Thread 1



Run\_queue



# Work stealing

Thread 2

Run\_queue



Thread N

Run\_queue



# Tokio IO

Pending Tasks



Thread 1



Run\_queue



Pending Tasks



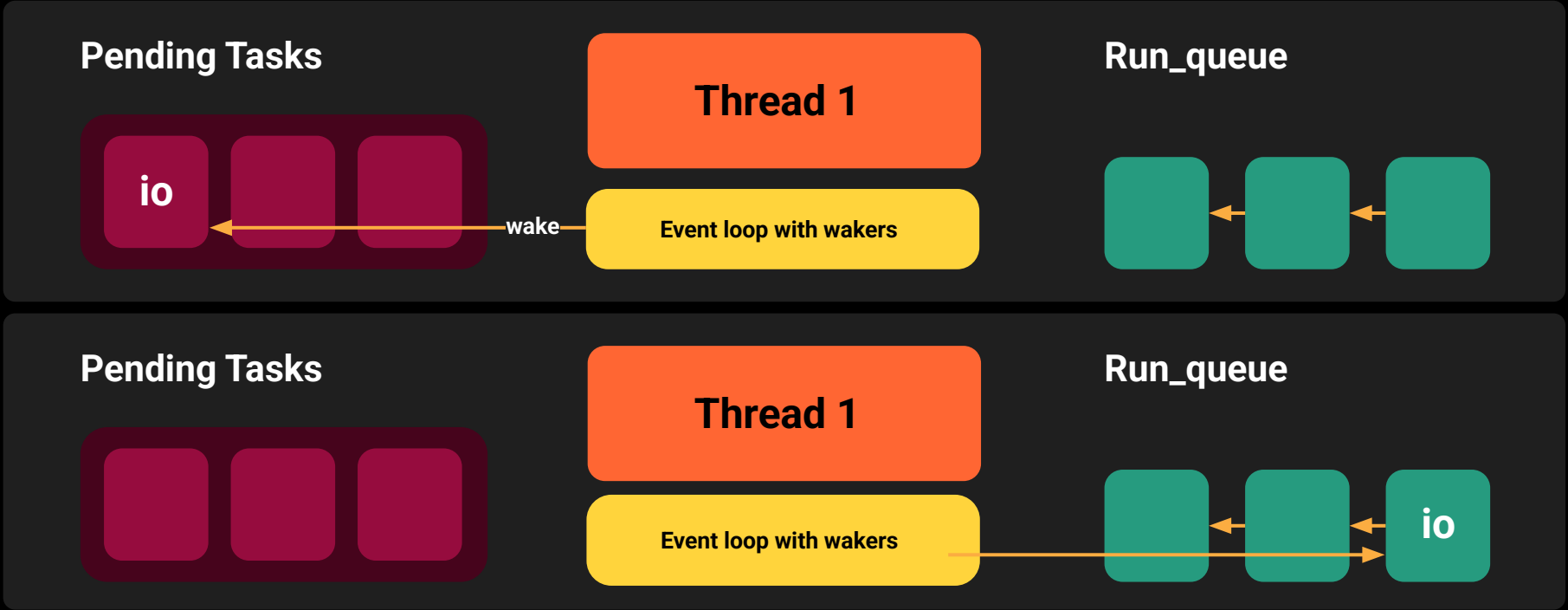
Thread 1



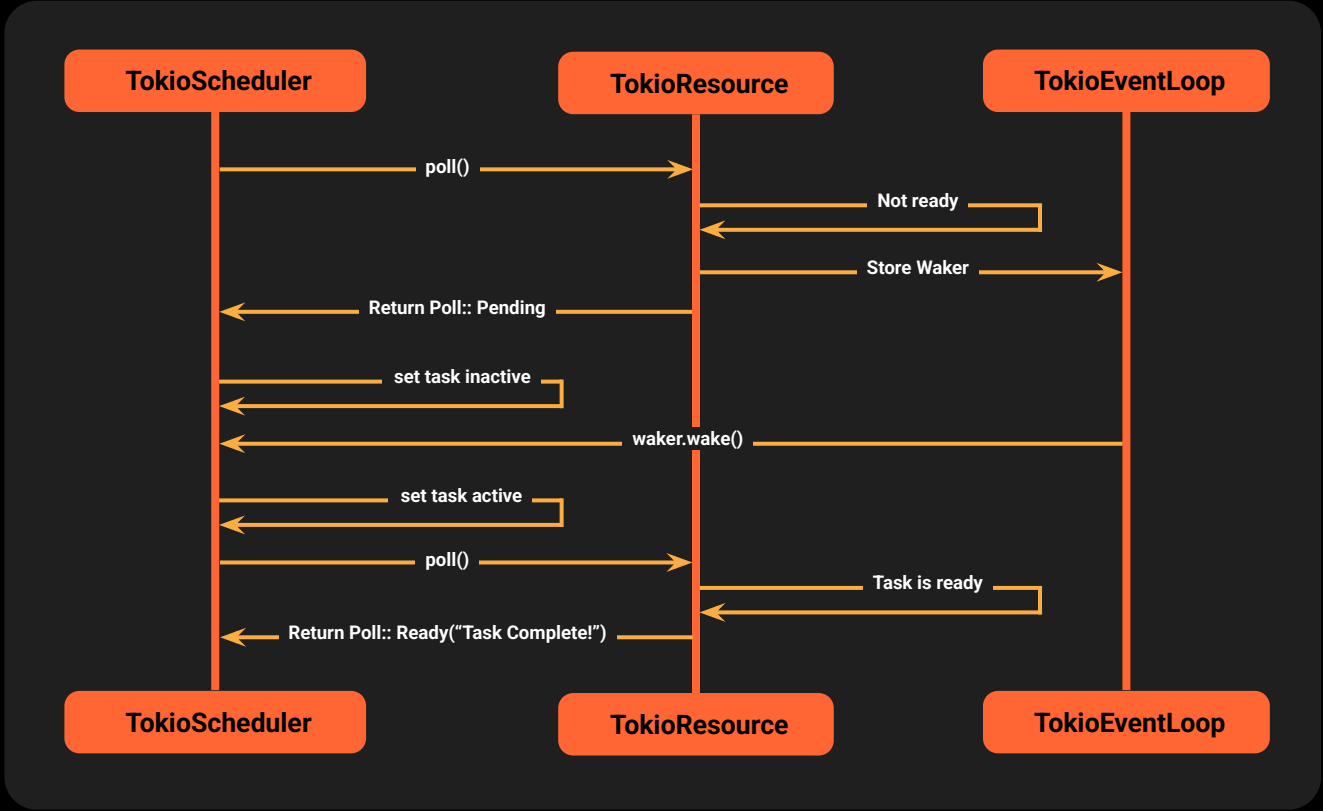
Run\_queue



# Tokio Resource



# Tokio Resource

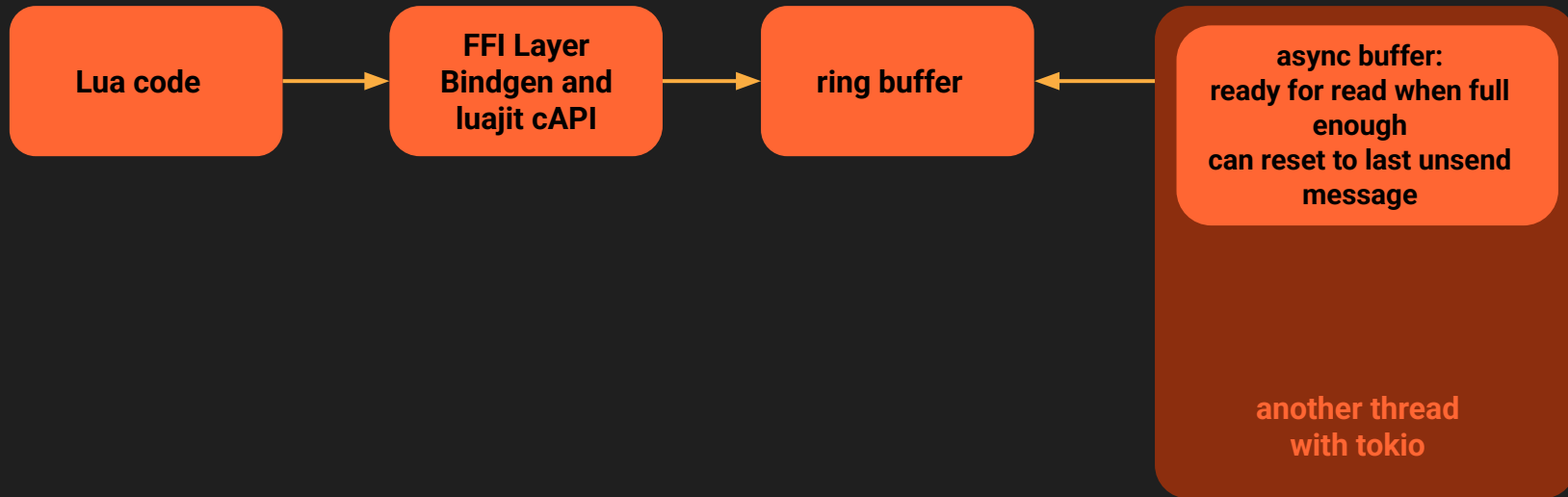


# Первое решение

# Что важно

- 1 Данные нужно слать батчами: сисколы не бесплатные
- 2 Нужно высылать данные периодически
- 3 В случае ошибок нужно переоткрывать соединение и пересылать данные с последнего недошедшего сообщения
- 4 Backpressure

# Что сделали



# Что сделали

```
pub async fn worker<T: SocketBuilder>(&mut self, socket_builder: &T) {
    let mut stream: impl AsyncWrite = loop {
        tokio::select! {
            conn = establish_connection:::<T::Output>(socket_builder, &self.socket_addr, &self.connection_retry_interval) => {break conn},
            _ = self.exit.recv(), if self.cons.cons.is_empty() => {return},
        }
    };

    let mut periodic_flush: Interval = interval(period: self.periodic_flush_interval);

    loop {
        tokio::select! {
            biased;
            res = copy_buf(&mut self.cons, &mut stream) =>
            {
                match res {
                    OK(_) => {
                        eprintln!("logger: finishing socket task");
                        return;
                    }
                    Err(e) => {
                        match e.kind() {
                            io::ErrorKind::WouldBlock | io::ErrorKind::Interrupted => {},
                            _ => {
                                eprintln!("logger: error writing into socket: {}", e);
                                stream = establish_connection(socket_builder, &self.socket_addr, &self.connection_retry_interval).await;
                                self.cons.reset_inflight_message();
                            }
                        }
                    },
                }
            },
            _ = periodic_flush.tick() => {
                self.cons.flush = true;
                self.wake();
            },
            _ = self.exit.recv() => {
                self.cons.exit = true;
                self.wake();
            }
        }; tokio::select!
    }
}

fn worker
```

# Что сделали

```
pub async fn worker<T: SocketBuilder>(&mut self, socket_builder: &T) {
    let mut stream: impl AsyncWrite = loop {
        tokio::select! {
            conn = establish_connection:::<T::Output>(socket_builder, &self.socket_addr, &self.connection_retry_interval) => {break conn},
            _ = self.exit.recv(), if self.cons.cons.is_empty() => {return,
        }
    };
    let mut periodic_flush: Interval = interval(period: self.periodic_flush_interval);
    loop {
        tokio::select! {
            biased;
            res = copy_buf(&mut self.cons, &mut stream) =>
            {
                match res {
                    OK(_) => {
                        eprintln!("logger: finishing socket task");
                        return;
                    }
                    Err(e) => {
                        match e.kind() {
                            io::ErrorKind::WouldBlock | io::ErrorKind::Interrupted => {},
                            _ => {
                                eprintln!("logger: error writing into socket: {}", e);
                                stream = establish_connection(socket_builder, &self.socket_addr, &self.connection_retry_interval).await;
                                self.cons.reset_inflight_message();
                            }
                        }
                    },
                }
            },
            _ = periodic_flush.tick() => {
                self.cons.flush = true;
                self.wake();
            },
            _ = self.exit.recv() => {
                self.cons.exit = true;
                self.wake();
            }
        }; tokio::select!
    }
}

fn worker
```

# Что сделали

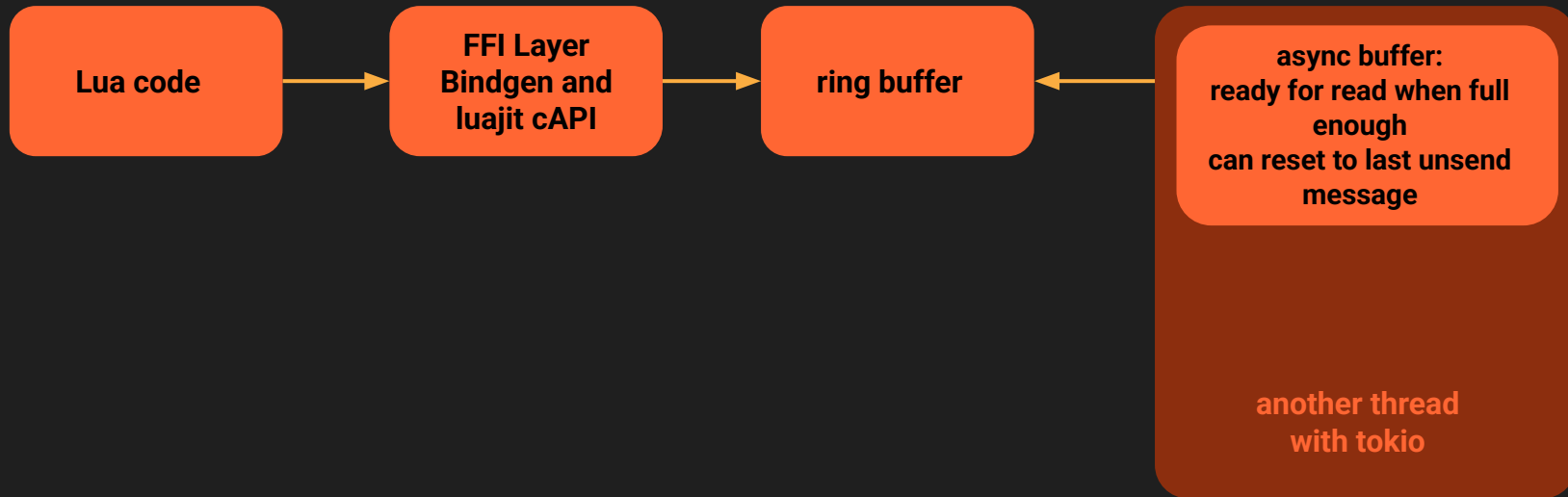
```
pub async fn worker<T: SocketBuilder>(&mut self, socket_builder: &T) {
    let mut stream: impl AsyncWrite = loop {
        tokio::select! {
            conn = establish_connection:::<T::Output>(socket_builder, &self.socket_addr, &self.connection_retry_interval) => {break conn},
            _ = self.exit.recv(), if self.cons.cons.is_empty() => {return,
            }
        };
        let mut periodic_flush: Interval = interval(period: self.periodic_flush_interval);
        loop {
            tokio::select! {
                biased;
                res = copy_buf(&mut self.cons, &mut stream) =>
                {
                    match res {
                        OK(_) => {
                            eprintln!("logger: finishing socket task");
                            return;
                        }
                        Err(e) => {
                            match e.kind() {
                                io::ErrorKind::WouldBlock | io::ErrorKind::Interrupted => {},
                                _ => {
                                    eprintln!("logger: error writing into socket: {}", e);
                                    stream = establish_connection(socket_builder, &self.socket_addr, &self.connection_retry_interval).await;
                                    self.cons.reset_inflight_message();
                                }
                            }
                        }
                    }
                }
            };
            _ = periodic_flush.tick() => {
                self.cons.flush = true;
                self.wake();
            },
            _ = self.exit.recv() => {
                self.cons.exit = true;
                self.wake();
            }
        }; tokio::select!
    }
}

fn worker
```

# Что сделали

```
pub async fn worker<T: SocketBuilder>(&mut self, socket_builder: &T) {
    let mut stream: impl AsyncWrite = loop {
        tokio::select! {
            conn = establish_connection:::<T::Output>(socket_builder, &self.socket_addr, &self.connection_retry_interval) => {break conn},
            _ = self.exit.recv(), if self.cons.cons.is_empty() => {return,
        }
    };
    let mut periodic_flush: Interval = interval(period: self.periodic_flush_interval);
    loop {
        tokio::select! {
            biased;
            res = copy_buf(&mut self.cons, &mut stream) =>
            {
                match res {
                    OK(_) => {
                        eprintln!("logger: finishing socket task");
                        return;
                    }
                    Err(e) => {
                        match e.kind() {
                            io::ErrorKind::WouldBlock | io::ErrorKind::Interrupted => {},
                            _ => {
                                eprintln!("logger: error writing into socket: {}", e);
                                stream = establish_connection(socket_builder, &self.socket_addr, &self.connection_retry_interval).await;
                                self.cons.reset_inflight_message();
                            }
                        }
                    },
                }
            }
            _ = periodic_flush.tick() => {
                self.cons.flush = true;
                self.wake();
            },
            _ = self.exit.recv() => {
                self.cons.exit = true;
                self.wake();
            }
        }; tokio::select!
    }
}
```

# Что сделали



# А минусы будут?



Копирование из кольцевого буфера в буфер для tokio!

# Синхронизация в многопоточном коде

# Синхронизация

**Поток 1**

Мутируем общий  
объект

**Общий объект**

```
Struct SharedObject  
&SharedObject  
Rc<SharedObject>
```

**Поток 2**

Мутируем общий  
объект

# Mutex<T>

```
▶ Run | Debug
1 fn main() {
2     let a: &Mutex<i32> = &std::sync::Mutex::new(10);
3
4     std::thread::scope(|s: &Scope<'_, '_>| {
5         s.spawn(|| {
6             let mut exclusive: MutexGuard<'_, i32> = a.lock().unwrap();
7             *exclusive = 20;
8             println!("{exclusive}");
9         });
10        s.spawn(|| {
11            let mut exclusive: MutexGuard<'_, i32> = a.lock().unwrap();
12            *exclusive = 30;
13            println!("{exclusive}");
14        });
15    });
16 }
```

# tokio::sync::Mutex<T>

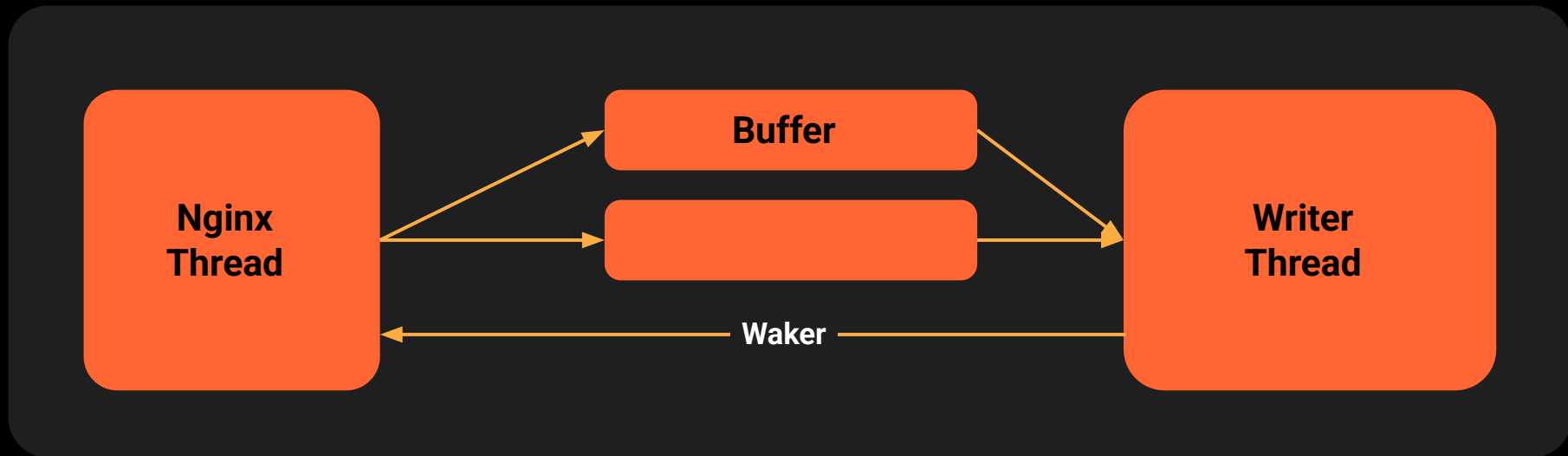
```
41 | async fn use_tokio_mutex() {  
42 |     let mtx: Mutex<i32> = tokio::sync::Mutex::new(0);  
43 |     let mut data: MutexGuard<'_, i32> = mtx.lock().await;  
44 |     inner_func().await;  
45 |     *data += 1;  
46 | }
```

# Atomic

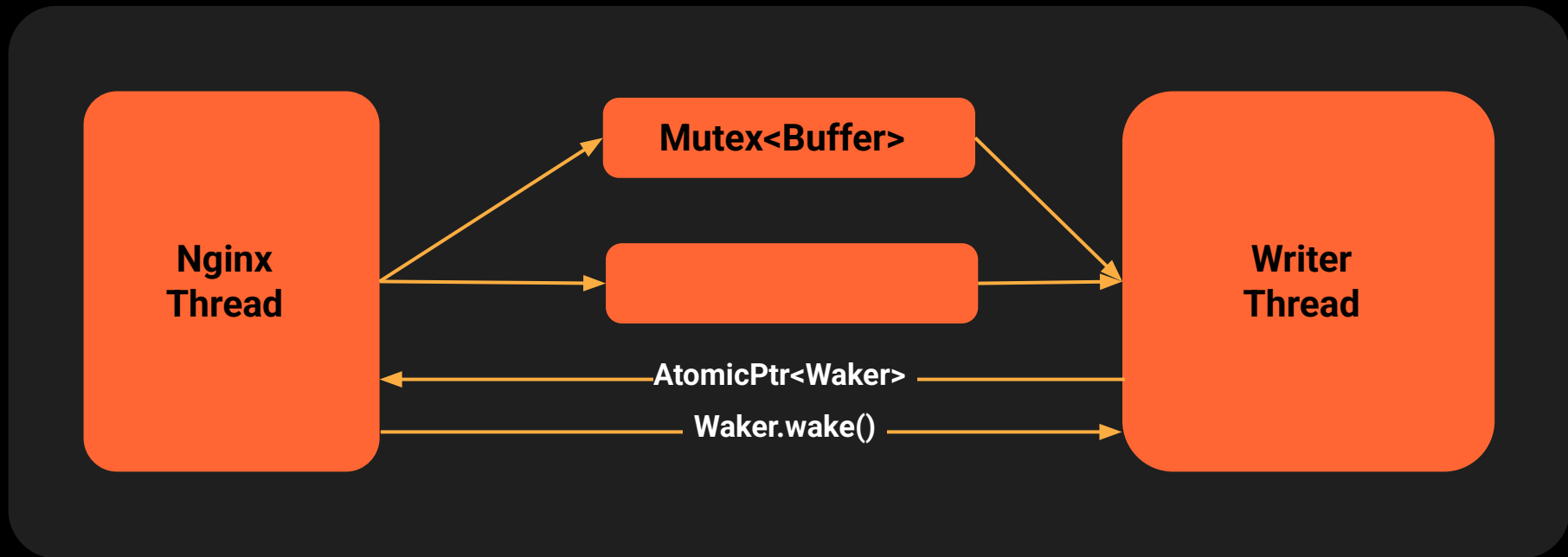
```
1  ∨ use core::sync::atomic::Ordering::Relaxed;
2    use core::sync::atomic::AtomicUsize;
3
4    ▶ Run | Debug
5  ∨ fn main() {
6      let a: &AtomicUsize = &AtomicUsize::new(10);
7
8  ∨  std::thread::scope(|s: &Scope<'_, '_>| {
9      ∨      s.spawn(|| {
10         a.store(val: 20, order: Relaxed);
11         println!("{}", a.load(Relaxed));
12     });
13     ∨      s.spawn(|| {
14         a.store(val: 30, order: Relaxed);
15         println!("{}", a.load(Relaxed));
16     });
17 }
```

# Улучшаем решение

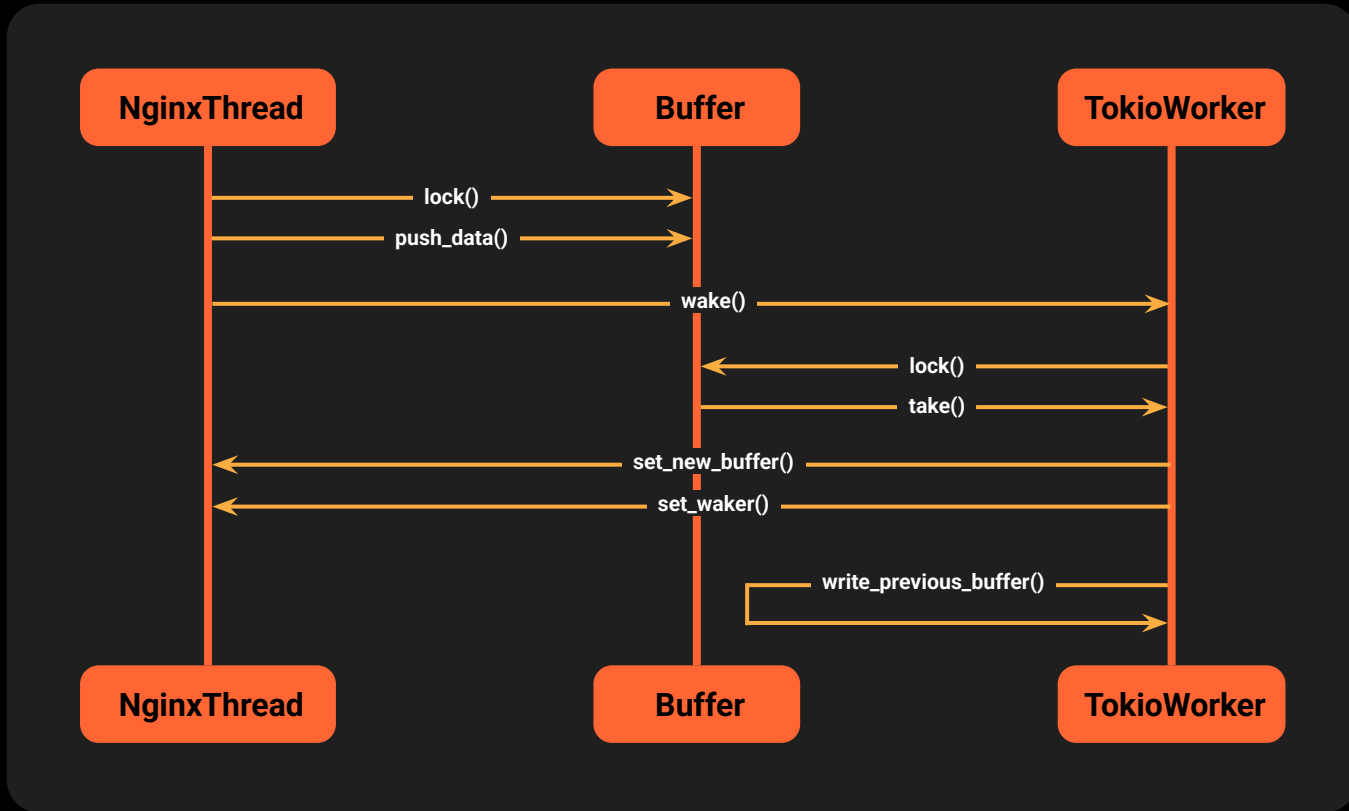
# Вторая версия



# Вторая версия



# Вторая версия



# Второе решение

```

103 ..... // wait for data to write
104 ..... if write_buffer.is_none() {
105 .....     write_buffer = Some(match self.shared_state.wait_and_fetch_buffer().await {
106 .....         Some(w: SealedBuffer) => w,
107 .....         None => {
108 .....             if self.shared_state.is_exiting() {
109 .....                 break; // You, 12 months ago • FLPLAT-3745: Simplify the main socket thread lo...
110 .....             }
111 .....             continue;
112 .....         }
113 .....     });
114 .....     socket_writer::buffer_await_seconds() Histogram
115 .....     .observe(self.received_data_time.elapsed().as_secs_f64());
116 .....     self.received_data_time = Instant::now();
117 ..... }
118 ..... // write to data to socket
0119 ..... match copy_buf(
120 .....     reader: &mut write_buffer.as_mut().unwrap(),
3122 .....     writer: &mut socket_stream.as_mut().unwrap(),
4123 .....     ) impl Future<Output = Result<, ...>
5124 .....     .await
6125 .....     {
7126 .....         Ok(_) => {
8127 .....             if let Some(ts: Duration) = write_buffer.unwrap().time_from_first_message() {
9128 .....                 socket_writer::message_processing_seconds().observe(ts.as_secs_f64());
0129 .....             }
1130 .....             write_buffer = None;
2131 .....         }
3132 .....         Err(e: Error) => {
4133 .....             match e.kind() {
5134 .....                 | io::ErrorKind::WouldBlock
6135 .....                 | io::ErrorKind::Interrupted
7136 .....                 | io::ErrorKind::WriteZero => {
8137 .....                 socket_writer::errors(error_message: &format!("{}", e.kind())).inc();
9138 .....             }
0139 .....             => {
1140 .....                 socket_writer::errors(error_message: &format!("{}", e.kind())).inc();
2141 .....                 socket_stream = None; // drop the connection
3142 .....                 write_buffer.as_mut().unwrap().reset();
4143 .....             }
5144 .....         }
6145 .....     }
7146 ..... };

```

# Второе решение (мьютекс)

```
pub async fn wait_and_fetch_buffer(&self) -> Option<SealedBuffer> {  
    let _ = timeout(duration: self.flush_interval, future: WaitForBuffer(self)).await;  
  
    let new_buf: IncomingMessagesBuffer = IncomingMessagesBuffer::new(flush_limit: self.flush_limit * 120 / 100); // use 1.2 coefficient to ensure minimum allocati  
    let buf: IncomingMessagesBuffer = { std::mem::replace(dest: &mut *self.messages_buffer.lock().unwrap(), src: new_buf) };  
  
    if !buf.as_slice().is_empty() {  
        return Some(SealedBuffer::new(buffer: buf));  
    }  
  
    None  
}
```

**А может быть  
попробуем spinlock?**

**А может быть  
попробуем spinlock?**



# Второе решение (atomic)

```

1 Implementation
struct WaitForBuffer<a>(&a SharedState);

impl<a> Future for WaitForBuffer<a> {
    type Output = ();

    fn poll(self: Pin<&mut Self>, cx: &mut Context<'a>) -> Poll<Self::Output> {
        let me: &mut WaitForBuffer<a> = self.get_mut();
        let previous_waker: *mut Waker = me.0.waker.swap(
            ptr: Box::into_raw(Box::new(cx.waker().clone())),
            order: Ordering::Acquire,
        );

        // previous_waker can be null only if awakened by writer thread;
        // it will be null for the first call but it doesn't matter if
        // we once immediately return not fully filled or empty buffer on start
        let previous_waker: *mut Waker
            .is_null() bool
            .then_some(Poll::Ready(())) Option<Poll<()>>
            .unwrap_or_else(|| {
                drop(unsafe { Box::from_raw(previous_waker) });
                if me.0.exit.load(order: Ordering::Relaxed) {
                    Poll::Ready()
                } else {
                    Poll::Pending
                }
            })
        };
    }
}
impl Future for WaitForBuffer<a>

```

```

fn try_wake(&self) {
    let ptr: *mut Waker = self.waker.swap(ptr: std::ptr::null_mut(), order: Ordering::Release);
    if !ptr.is_null() {
        // This is just sending a hint for tokio thread object, no actual swapping performed;
        // Tokio-thread with socket decides independently when it wants to fetch the buffer.
        // Competes for lock with update_waker from tokio-thread, which sets the waker object if it is needed;
        // Again lock expected to be free most of the time.
        unsafe { Box::from_raw(ptr) }.wake();
    }
}

```

# Про тестирование

# Тестирование

## Mockall

```
8 mock! {
9     pub Socket {}
10    impl AsyncWrite for Socket {
11        fn poll_write<'a>(
12            self: Pin<&mut Self>,
13            cx: &mut Context<'a>,
14            buf: &[u8]
15        ) -> Poll<Result<usize, io::Error>>;
16        fn poll_flush<'a>(
17            self: Pin<&mut Self>,
18            cx: &mut Context<'a>
19        ) -> Poll<Result<(), io::Error>>;
20        fn poll_shutdown<'a>(
21            self: Pin<&mut Self>,
22            cx: &mut Context<'a>
23        ) -> Poll<Result<(), io::Error>>;
24    }
25 }

26
27 mock! {
28     pub ConnectionBuilder {}
29     impl SocketBuilder for ConnectionBuilder {
30         type Output = MockSocket;
31         fn build(&self) -> impl Future<Output = io::Result<MockSocket>>;
32     }
33 }
```

# Тестирование

```
605 | ..... | ..... | ..... | ..... | let mut connection_builder: MockConnectionBuilder = MockConnectionBuilder::new();
606 | ..... | ..... | ..... | ..... | connections Arc<Mutex<Vec<Arc<Mutex<Vec<...>>>>>>
607 | ..... | ..... | ..... | ..... |     .lock() Result<MutexGuard<'_, Vec<...>>, ...>
608 | ..... | ..... | ..... | ..... |     .unwrap() MutexGuard<'_, Vec<Arc<Mutex<...>>>>
609 | ..... | ..... | ..... | ..... |     .push(mock_unstable_connection(&mut connection_builder));
610 | ..... | ..... | ..... | ..... | connection_builder
```

# Тестирование

```
515  ✓  ... fn mock_unstable_connection(  
516  ...  ... connection_builder: &mut MockConnectionBuilder,  
517  ✓  ... ) -> Arc<Mutex<Vec<u8>>> {  
518  ✓  ...  ... const CONNECTION_ERROR_PERCENT: usize = 10;  
519  ...  ... const SOCKET_WRITE_ERROR_PERCENT: usize = 10;  
520  ...  ... const SOCKET_OVERLOADED_ERROR_PERCENT: usize = 10;  
521  ...  ... const SOCKET_PENDING_PERCENT: usize = 10;  
522  ...  ... const SOCKET_WAIT_ON_PENDING: std::time::Duration = std::time::Duration::from_millis(1);  
523  
524  ...  ... let received_buffer: Arc<Mutex<Vec<u8>>> = Arc::new(data: M ... );  
525  
526  ✓  ...  ... connection_builder.expect_build().returning(__mockall_f: {
```

Click to collapse the range.

# Тестирование

```
582     ... #[test]
583     ...     ▶ Run Test | ⌘ Debug
584     ...     fn load_test_with_random_errors() {
585     ...         use rand;
586     ...         const AVG_MESSAGE_SIZE: usize = 4 * 1024;
587     ...         const POOL_SIZE: usize = 4;
588     ...
589     ...         let mut conf: HashMap<&str, usize> = HashMap::new();
590     ...         conf.insert(k: "pool_size", v: POOL_SIZE);
591     ...         conf.insert(k: "drop_limit", v: AVG_MESSAGE_SIZE * 2 + 1);
592     ...         conf.insert(k: "flush_limit", v: AVG_MESSAGE_SIZE * 3);
593     ...         conf.insert(k: "max_buffer_length", v: AVG_MESSAGE_SIZE * 1024);
594     ...         conf.insert(k: "periodic_flush_interval", v: 3000);
595     ...         conf.insert(k: "connection_retry_interval", v: 100);
596     ...     }
```

**Так, а какие выводы?**

Вывод:

**Rust для таких  
сетевых  
задач — это ...**





**Денис Давыдов**  
Systems Engineer,  
Cloudflare, FL Team  
UK, Лондон



@andrushaTheSlayer



davidovdd92@mail.ru



<https://github.com/WowVeryLogin>