Rust 异步编程原理和实践

About me

- 方圆 （@fanngyuan）
- Samsung
- CISCO
- WEIBO
- 创业
- 罗辑思维
- Westar
  - https://github.com/starcoinorg/starcoin
- Github： https://github.com/fanngyuan

Starcoin 新一代Libra

Starcoin x Move

新的数字资产编程方式

- PoW
- Move
- 分层
- Stdlib
- 欢迎部署Move合约

Outline

- Rust network programming
- Futures
- Tokio
- Async/await
- Rust async@Starcoin

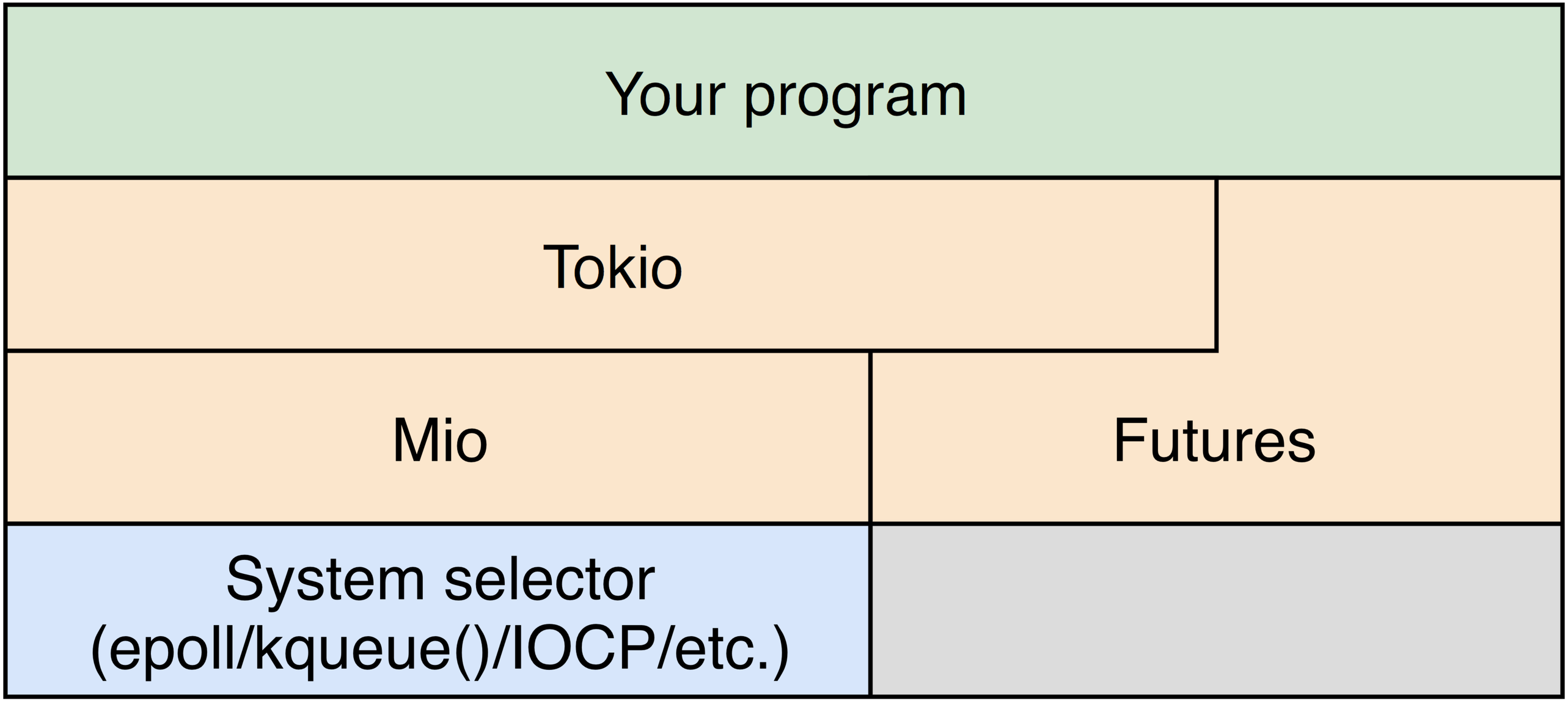
- Rust Network Programming Features

  - Block IO + Thread
  - NoBlocking IO + Callback
  - Coroutine

Rust Async Programming Features

- Future based coroutine
- Zero cost abstraction
- Fast
  - No runtime allocations
  - No dynamic dispatch
  - No gc
- Safety

Tokio and Rust Async



Futures

What’s future?

- Database Query
- Rpc
- ……

- pull , not push

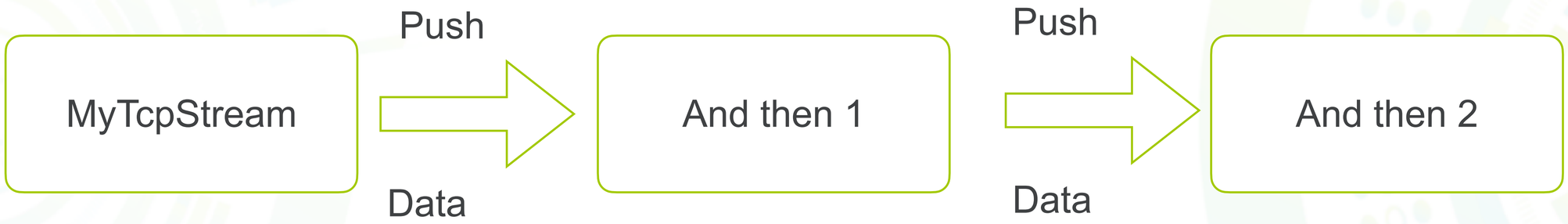
```
struct MyTcpStream {
  nread: u64,
  callback: Option<Box<Fn(u64)>>,
}

// this is push model
```

```
let f=MyTcpStream::connect(&remote_addr)
  .and_then(|sock| io::read_exact(sock, 10))
  .and_then(|(sock, response)| {
    process(response)
  });

tokio::spawn(f);
```

MyTcpStream

And then 1

And then 2


MyTcpStream
Push
Data
And then 1
Push
Data
And then 2

why not push?

- extra dynamic memory alloc
- control logic
- ……

Poll future

```rust
pub trait Future {

    type Item;

    type Error;


    fn poll(&mut self) -> Poll<Self::Item, Self::Error>;
}
```

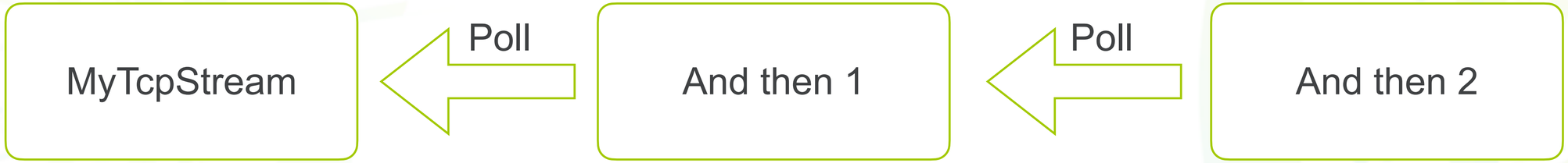
```rust
struct MyTcpStream {
    socket: TcpStream,
    nread: u64,
}

impl Future for MyTcpStream {
    type Item =u64;
    type Error = io::Error;

    fn poll(&mut self) -> Poll<Item, io::Error> {
        let mut buf = [0;10];
        loop {
            match self.socket.read(&mut buf) {
                Async::Ready(0) => return Async::Ready(self.nread),
                Async::Ready(n) => self.nread += n,
                Async::NotReady => return Async::NotReady,
            }
        }
    }

}
```

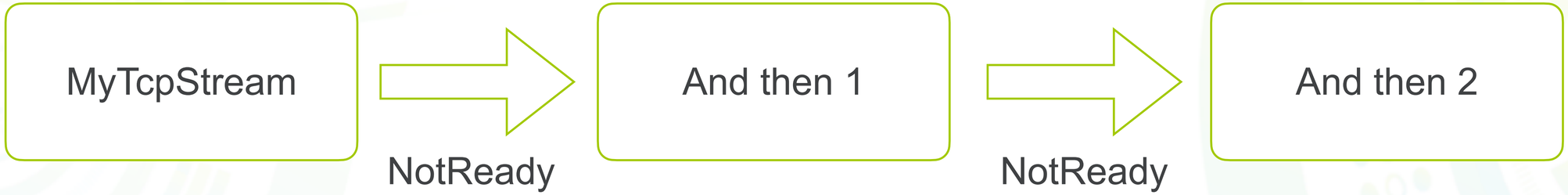
```rust
enum AndThen<A,F> {
    First(A, F),
}

fn poll(&mut self) -> Async<Item> {
    match fut_a.poll() {
        Async::Ready(v) => f(v),
        Async::NotReady => Async::NotReady,
    }
}
/// and then will move previous future
```

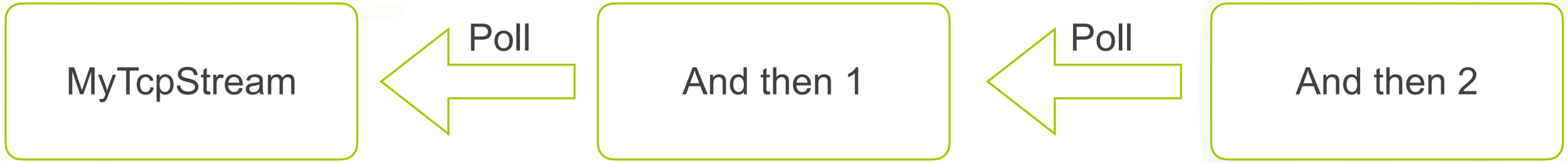
MyTcpStream

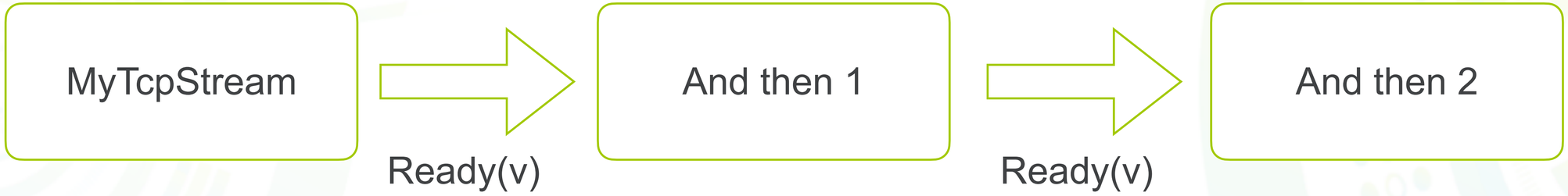
And then 1

And then 2


MyTcpStream
Poll
And then 1
Poll
And then 2


MyTcpStream
NotReady
And then 1
NotReady
And then 2


MyTcpStream
Poll
And then 1
Poll
And then 2

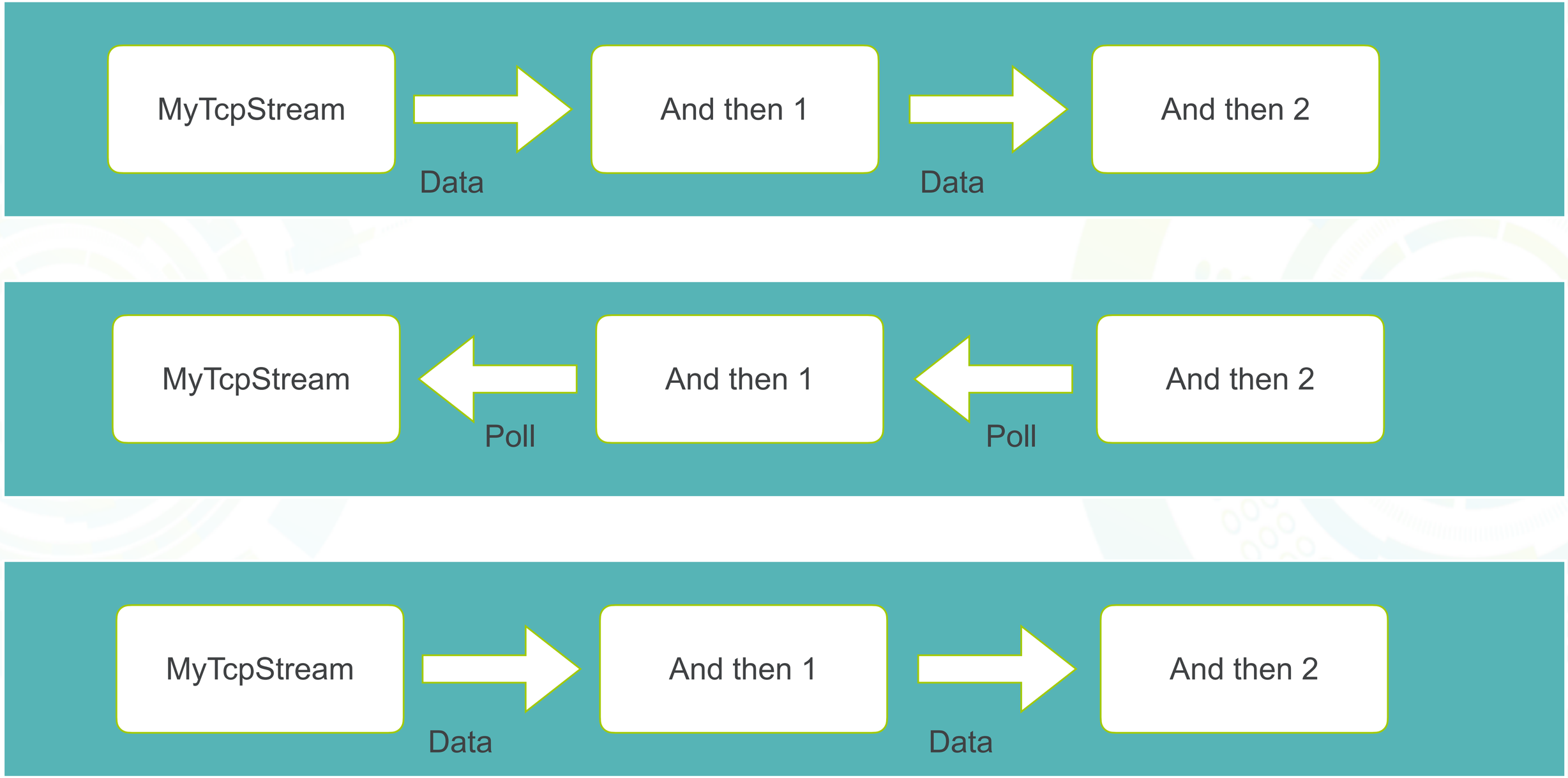

MyTcpStream
Ready(v)
And then 1
Ready(v)
And then 2

- connect to server
- send handshake
- read handshake response
- send request
- handle response

```
MyTcpStream::connect(&remote_addr)
  .and_then(|sock| io::write(sock, handshake))
  .and_then(|sock| io::read_exact(sock, 10))
  .and_then(|(sock, handshake)| {
    validate(handshake);
    io::write(sock, request)
  })
  .and_then(|sock| io::read_exact(sock, 10))
  .and_then(|(sock, response)| {
    process(response)
  })
```

```
tokio::spawn(future)
```

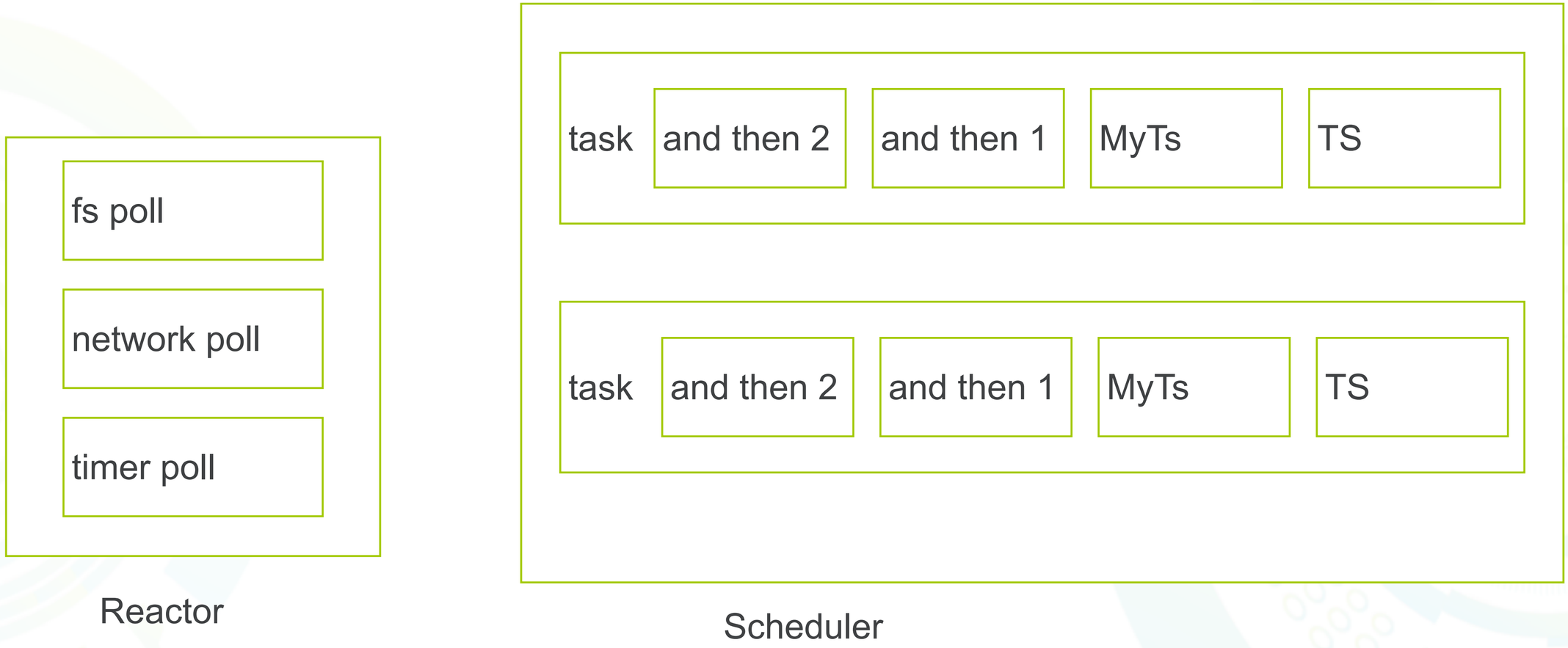
```rust
pub(crate) struct Task {
    state: AtomicUsize,
    blocking: AtomicUsize,
    next: AtomicPtr<Task>,
    next_blocking: AtomicPtr<Task>,
    future: UnsafeCell<Option<TaskFuture>>,
}
```

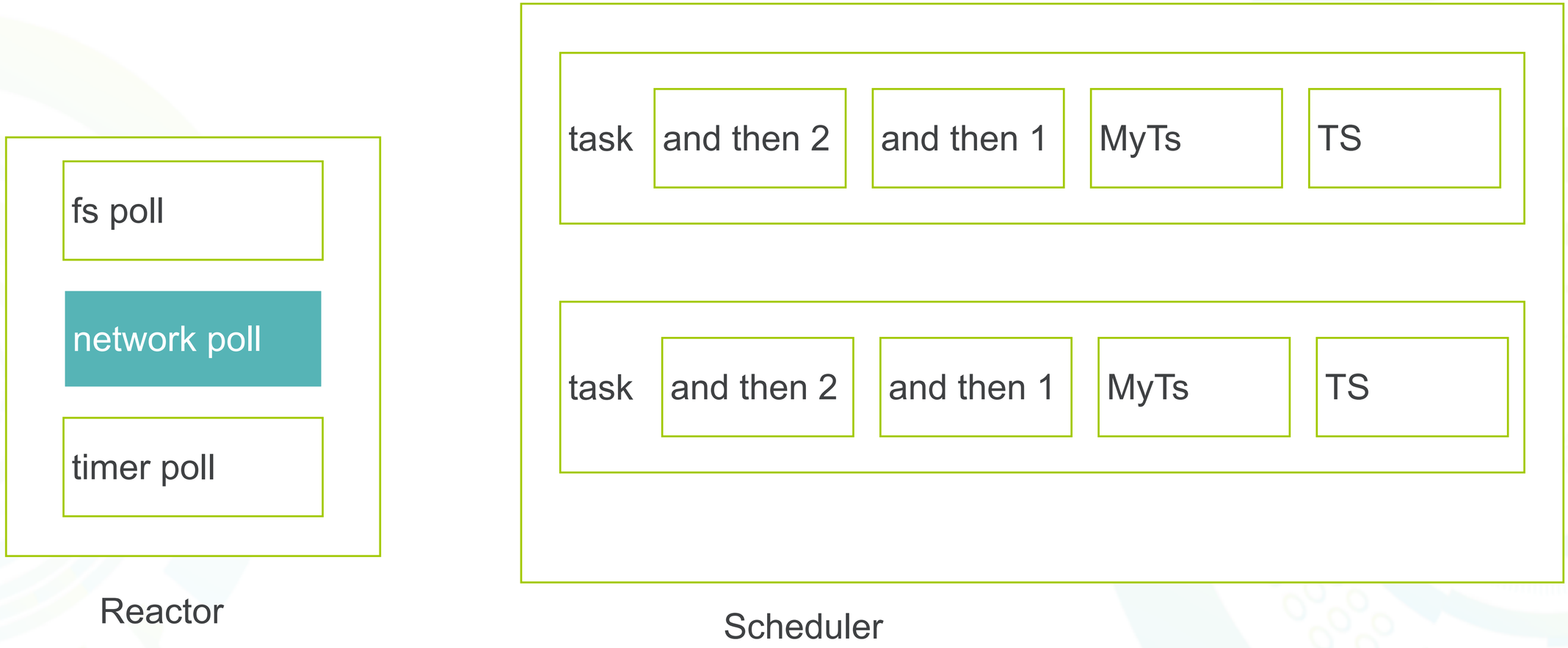

MyTcpStream
Data
And then 1
Data
And then 2
MyTcpStream
Poll
And then 1
Poll
And then 2
MyTcpStream
Data
And then 1
Data
And then 2

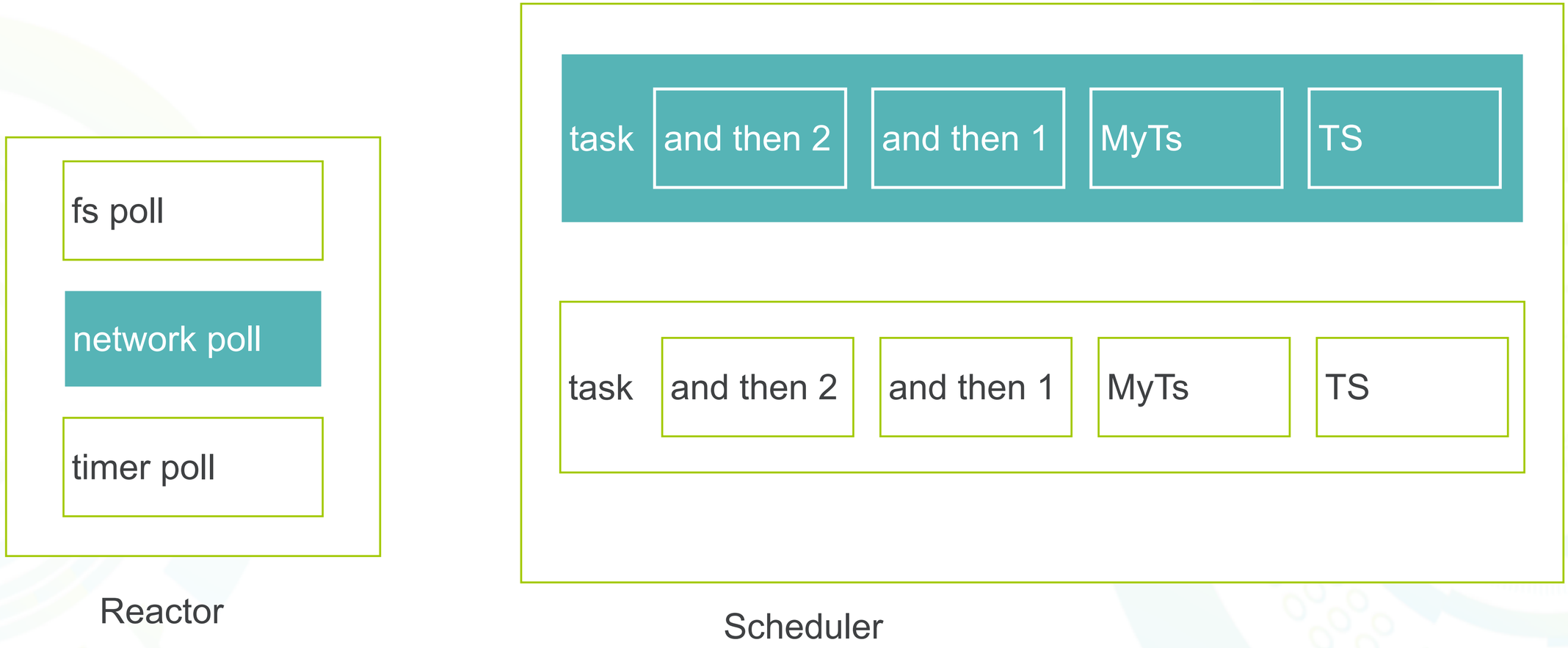
Tokio

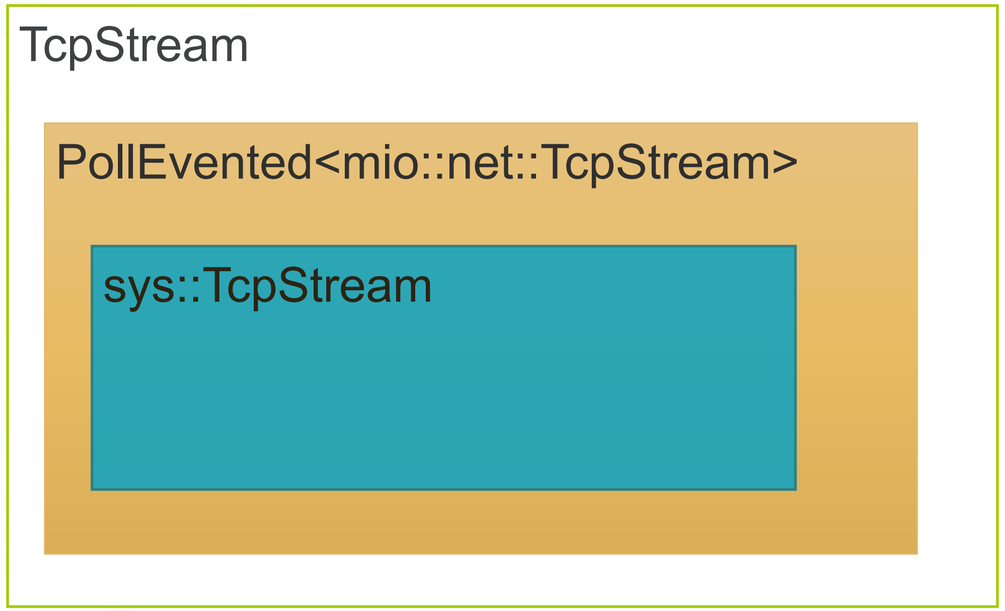
- based on Mio
  - Epoll,kqueue,IOCP
- Timers
- Task scheduling
- File System Access
- Others

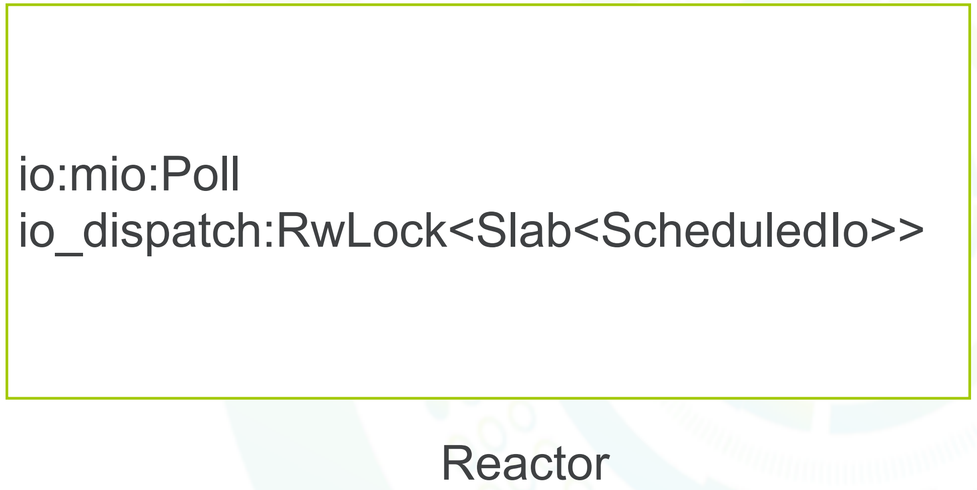
```rust
let listener = TcpListener::bind(&addr).unwrap();

let server = listener.incoming().for_each(move |socket| {
    tokio::spawn(process(socket));
    Ok(())
}).map_err(|err| {
        println!("accept error = {:?}", err);
});

tokio::run(server);
```


fs poll
network poll
timer poll
Reactor
task
and then 2
and then 1
MyTs
TS
task
and then 2
and then 1
MyTs
TS
Scheduler

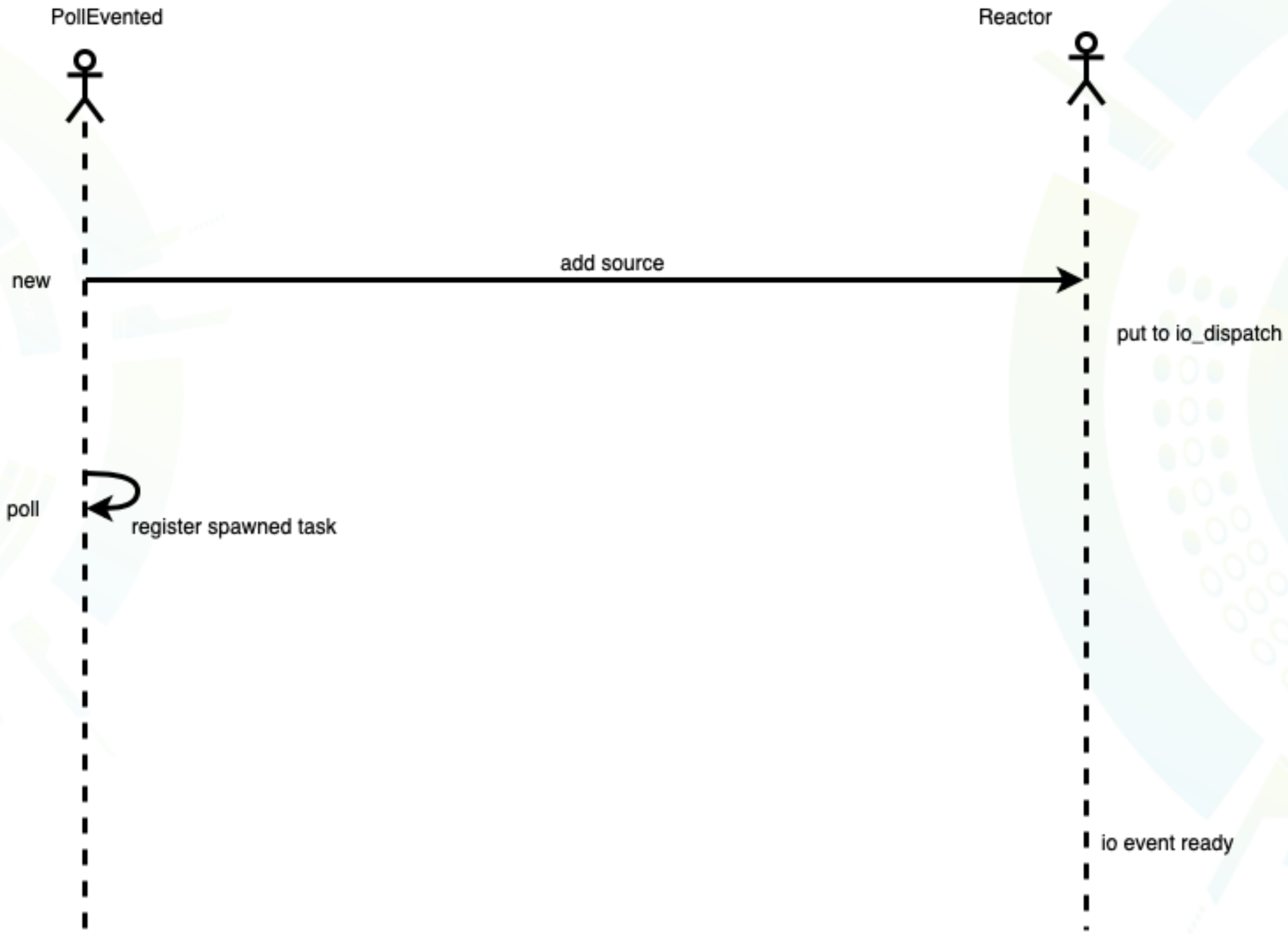

fs poll
network poll
timer poll
Reactor
task
and then 2
and then 1
MyTs
TS
task
and then 2
and then 1
MyTs
TS
Scheduler


fs poll
network poll
timer poll
Reactor
task
and then 2
and then 1
MyTs
TS
task
and then 2
and then 1
MyTs
TS
Scheduler


TcpStream
PollEvented<mio::net::TcpStream>
sys::TcpStream


io:mio:Poll
io_dispatch:RwLock<Slab<ScheduledIo>>

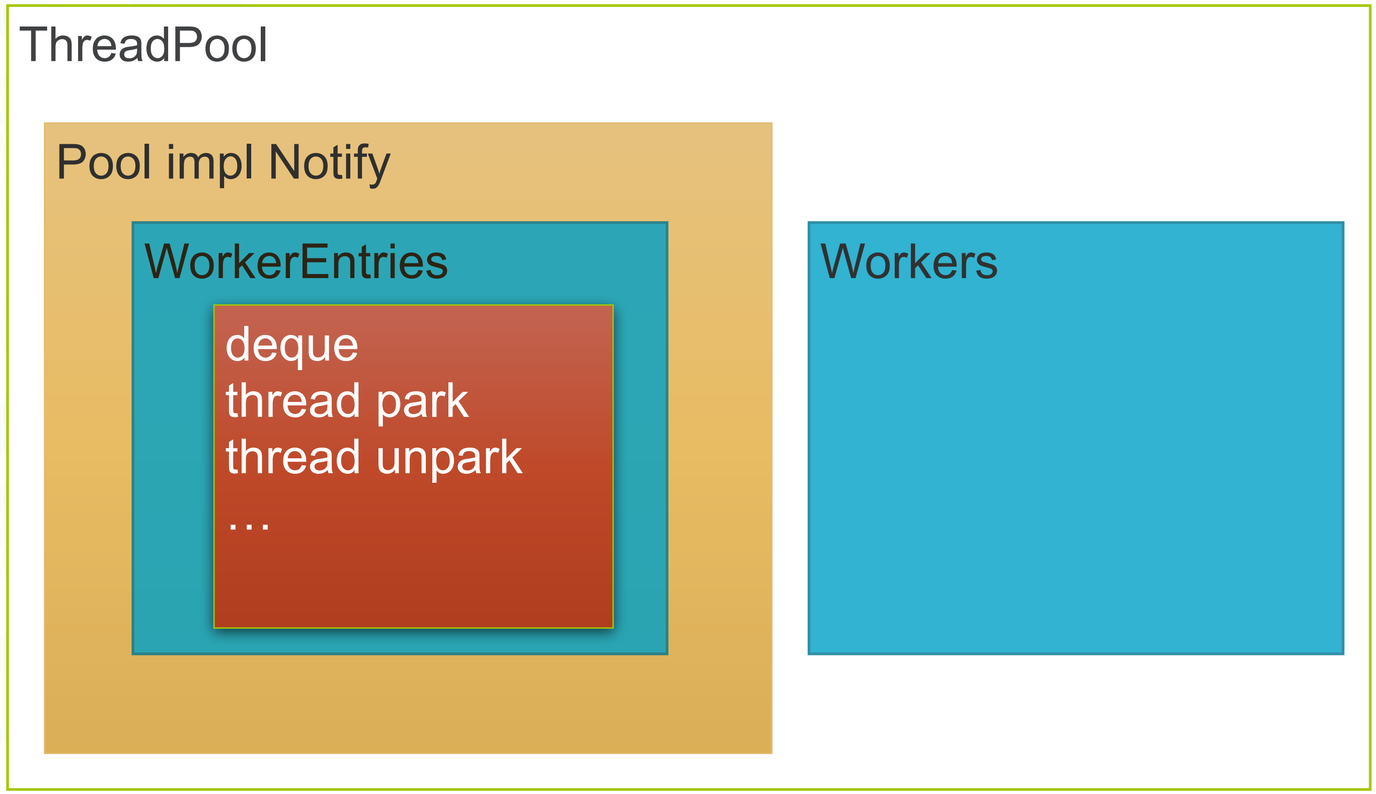
Reactor

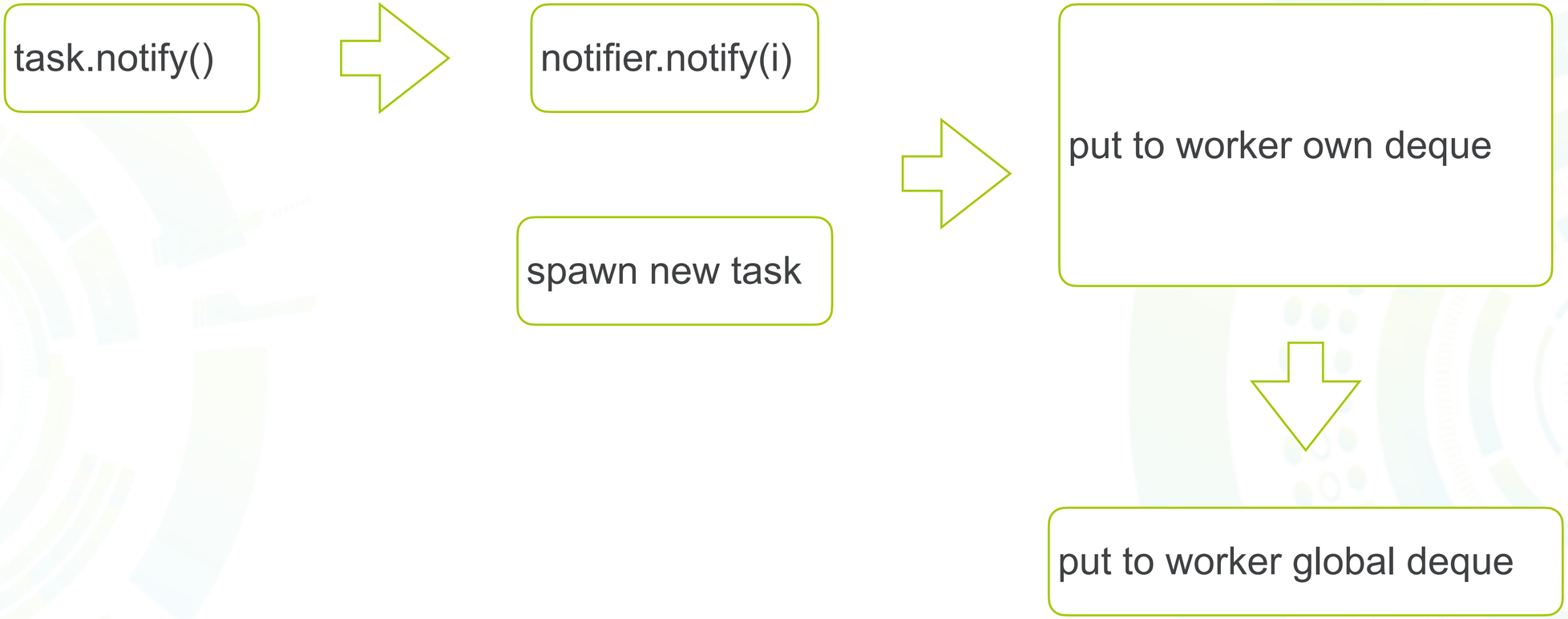

PollEvented
Reactor
new
add source
put to io_dispatch
poll
register spawned task
io event ready

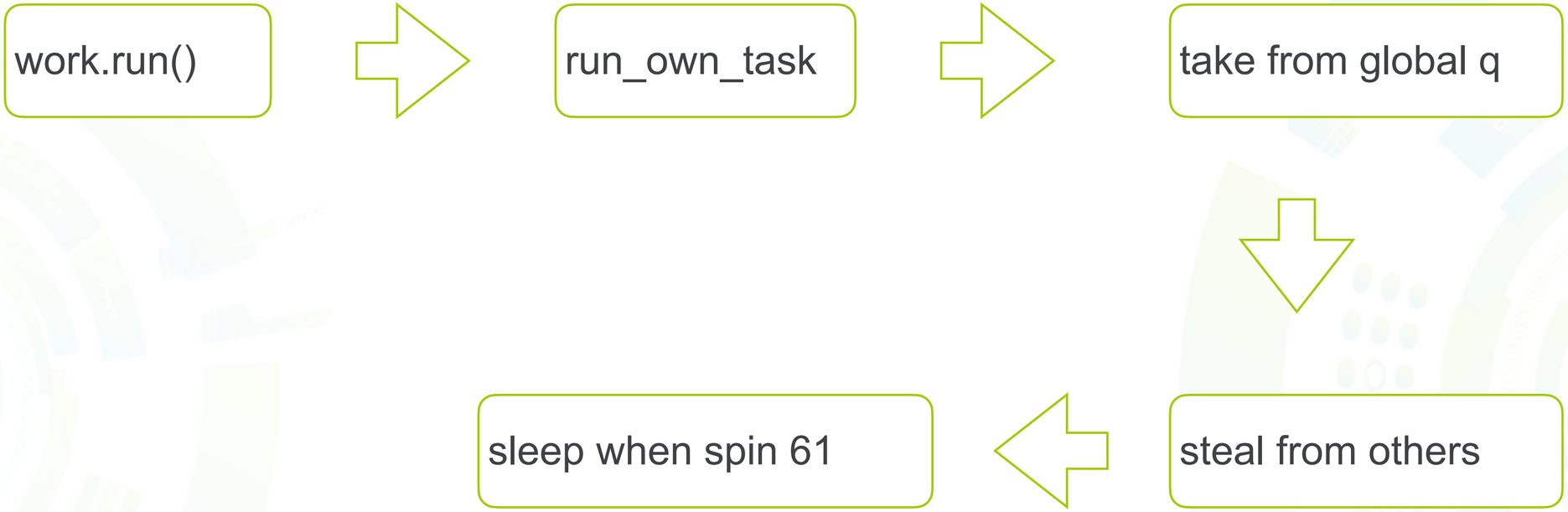
```
task::current() -> Task
Task::notify()
```

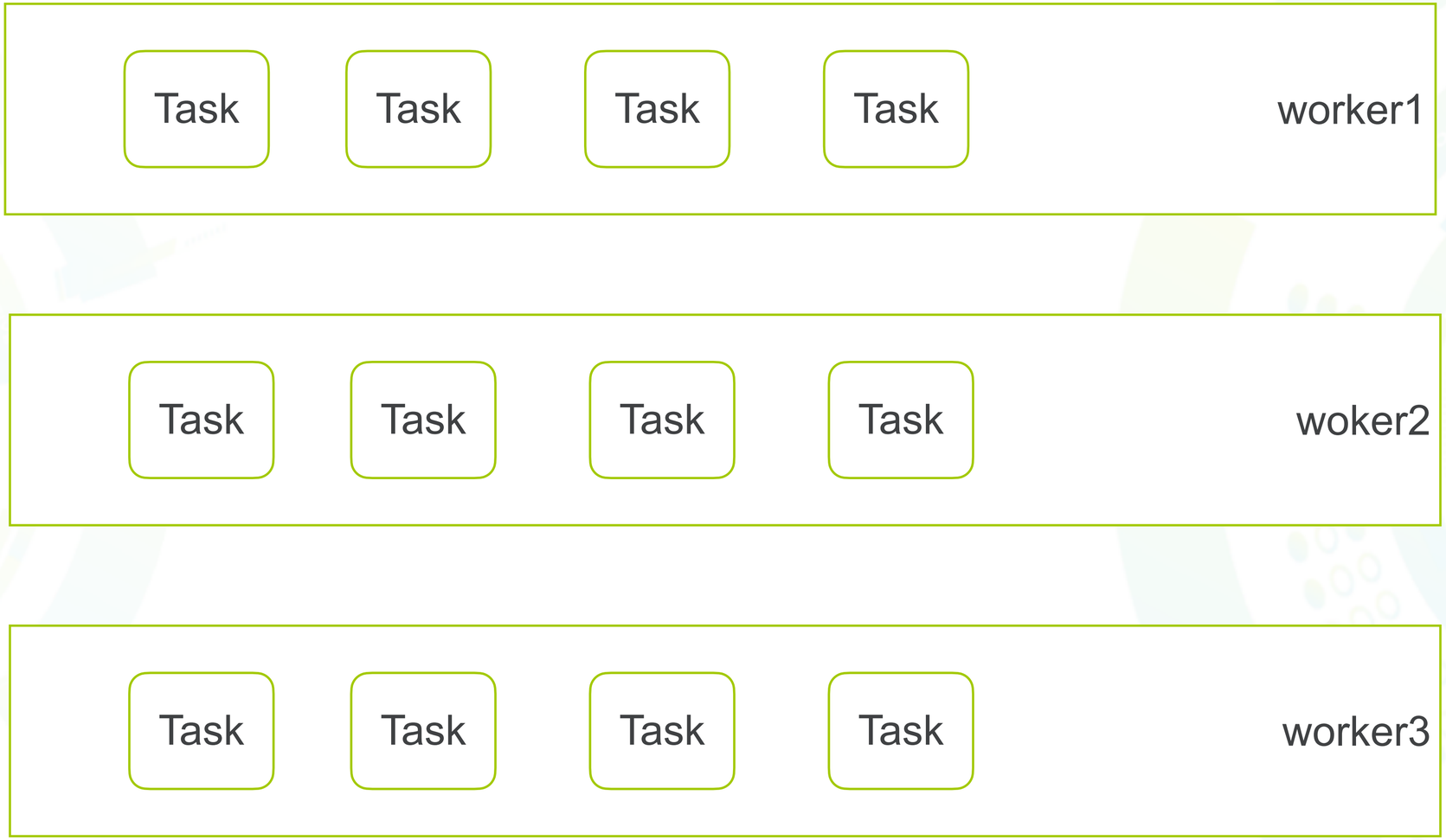
```rust
pub trait Notify: Send + Sync {

    fn notify(&self, id: usize);

    fn clone_id(&self, id: usize) -> usize {
        id
    }

    fn drop_id(&self, id: usize) {
        drop(id);
    }

}
// this trait need implemented by scheduler
```


tokio:run()/runtime::new()
threadpool::new()


ThreadPool
Pool impl Notify
WorkerEntries
deque
thread park
thread unpark
…
Workers


task.notify()
notifier.notify(i)
spawn new task
put to worker own deque
put to worker global deque


work.run()
run_own_task
take from global q
steal from others
sleep when spin 61


Task
Task
Task
Task
worker1
Task
Task
Task
Task
woker2
Task
Task
Task
Task
worker3

```rust
pub struct Waker {
    waker: RawWaker,
}

impl Waker {

    pub fn wake(self) {
    }

    pub fn wake_by_ref(&self) {
    }
}
```

async/await

```rust
#[tokio::main]
pub async fn main() -> Result<(), Box<dyn Error>> {
    let mut stream = TcpStream::connect("127.0.0.1:6142").await?;
    println!("created stream");
    let result = stream.write(b"hello world\n").await;
    println!("wrote to stream; success={:?}", result.is_ok());
    Ok(())
}
```

```rust
async fn f()-> i32 {
  //
}
```

```rust
fn f() -> impl Future<Output=i32>{
  //
}
```

```rust
async fn b()-> i32 {
  f().await
}
```

```rust
struct B {
   future_f:Option<FutureF>
}

impl Future for B {
    type Output = i32;

    fn poll(mut self: Pin<&mut Self>, cx: &mut Context) -> Poll<Self::Output> {
    loop {
            match self.future_f.take().expect("take").poll(tx) {
            ///
        }
    }
    }
}
```

```
async fn off_chain_pay_htlc(
        router:&Router,
        receiver_address: AccountAddress,
        amount: u64,
        hash_lock: Vec<u8>,
        timeout: u64,) -> Result<()> {
        let path = router.find_path_by_addr(self.wallet.account(), receiver_address).await?;
        pay_multi_hop_request(path, amount, hash_lock, timeout).await?
        Ok(())
}
```

```rust
struct HtlcHandle {
    receiver_address: AccountAddress,
    amount: u64,
    hash_lock: Vec<u8>,
    timeout: u64,
    router:Router,
    find_path_future:Option<Future<Output=X>>,
    pay_future:Option<Future<Output=()>>,
    state:State
}
```

```rust
enum State {
    Unpolled,
    GetPath,
    Pay(path),
    Ready,
}
```

```rust
impl Future for HtlcHandle {
    type Output = ();

    fn poll(mut self: Pin<&mut Self>, cx: &mut Context) -> Poll<Self::Output> {
        loop {
        match self.state {
        Unpolled => {
            // ...
            self.state=GetPath;
        }
        GetPath => {
            match self.find_path_future.poll(cx){
                //
            }
            self.state = Pay;
        }
        Pay => {
            match self.pay_future.poll(cx){
                //
            }
            self.state = Ready;
        }
        Ready => {
            return ();
        }
        }
        }
    }
}
```

Build your own async/await

```rust
#![feature(generators, generator_trait)]

use std::ops::{Generator, GeneratorState};
use std::pin::Pin;

fn main() {
    let mut generator = || {
        yield 1;
        return "foo"
    };

    match Pin::new(&mut generator).resume(()) {
        GeneratorState::Yielded(1) => {}
        _ => panic!("unexpected value from resume"),
    }
    match Pin::new(&mut generator).resume(()) {
        GeneratorState::Complete("foo") => {}
        _ => panic!("unexpected value from resume"),
    }
}
```

Build your own async/await

```
let xs = vec![1, 2, 3];
let mut gen = move|| {
    let mut sum = 0;
    for x in xs {
        sum += x;
        yield sum;
    }
};
```

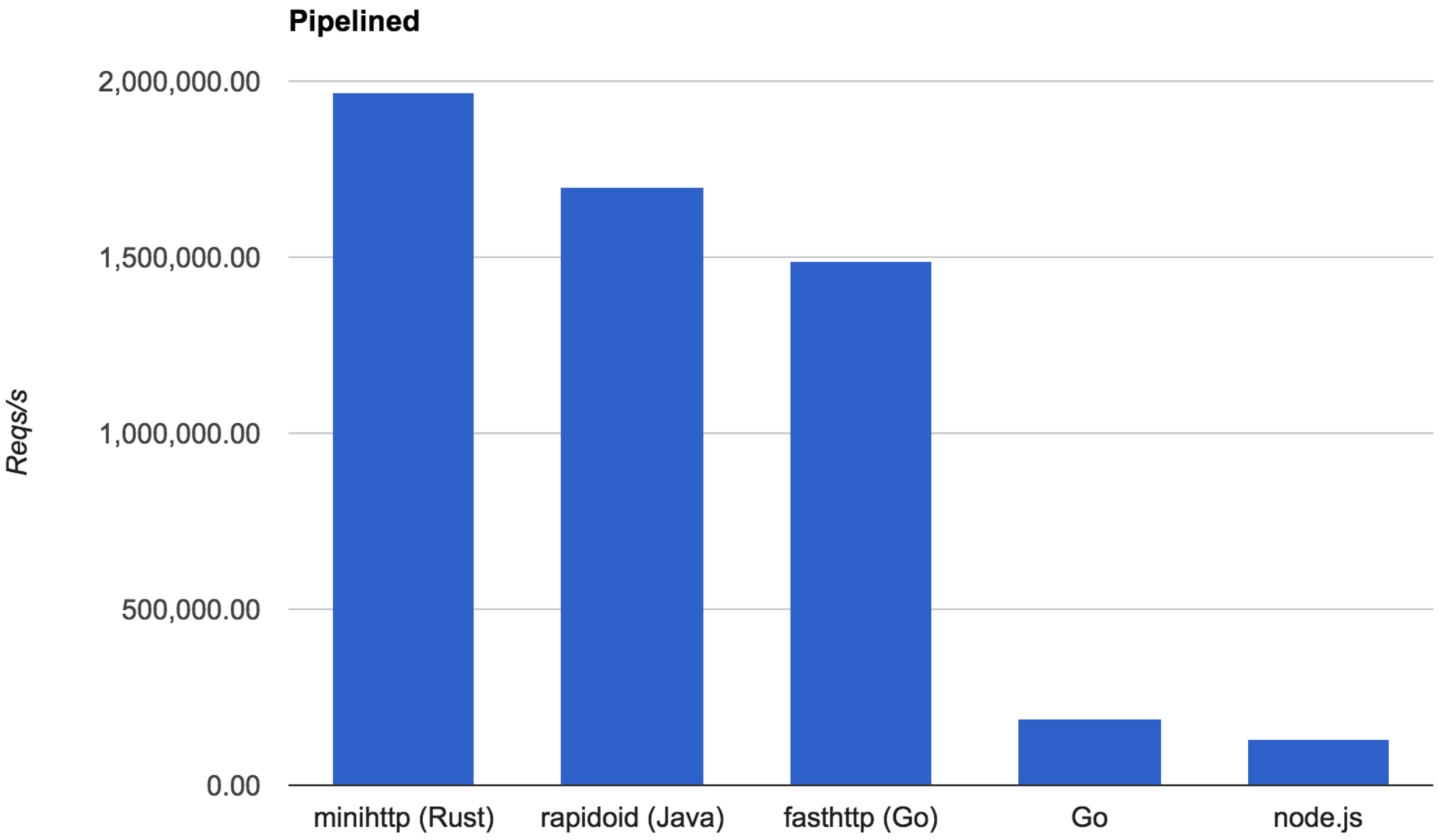
Build your own async/await

```rust
impl<T: Generator<Yield = ()>> Future for GenFuture<T> {
    type Output = T::Return;
    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Output> {
        // Safe because we're !Unpin + !Drop mapping to a ?Unpin value
        let gen = unsafe { Pin::map_unchecked_mut(self, |s| &mut s.0) };
        let _guard = unsafe { set_task_context(cx) };
        match gen.resume() {
            GeneratorState::Yielded(()) => Poll::Pending,
            GeneratorState::Complete(x) => Poll::Ready(x),
        }
    }
}
```

Build your own async/await

```
macro_rules! await_macro {
    ($e:expr) => ({
        let mut future = $e;
        loop {
            if let Poll::Ready(x) =
            poll_with_tls_context(unsafe { Pin::new_unchecked(&mut future) })
            {
                break x;
            }
            yield
        }
    })
}
```

Rust async@Starcoin


Pipelined
Reqs/s
2,000,000.00
1,500,000.00
1,000,000.00
500,000.00
0.00
minihttp (Rust)
rapidoid (Java)
fasthttp (Go)
Go
node.js

Problems

- life cycle
  - self
- futures and tokio
- notify task by your self
- futures01 and futures 03

Async in Starcoin

- Runtime
  - Different runtime for different business
- TaskExecutor（Handle）
  - tokio::spawn for default runtime
  - get from runtime
  - clone
- async/await
- Actor model

Actor

```rust
pub struct AntRouter {
    executor: Handle,
    command_sender: UnboundedSender<RouterCommand>,
    inner: Option<AntRouterInner>,
    network_receiver: Option<UnboundedReceiver<(AccountAddress, RouterNetworkMessage)>>,
    command_receiver: Option<UnboundedReceiver<RouterCommand>>,
    control_receiver: Option<UnboundedReceiver<Event>>,
    control_sender: UnboundedSender<Event>,
    stats_mgr: Arc<Stats>,
}

struct AntRouterInner {
    network_sender: UnboundedSender<(AccountAddress, RouterNetworkMessage)>,
    wallet: Arc<WalletHandle>,
    seed_manager: SeedManager,
    message_processor: MessageProcessor<RouterNetworkMessage>,
    default_future_timeout: AtomicU64,
    executor: Handle,
    stats_mgr: Arc<Stats>,
    path_store: PathStore,
}
```

Async in Starcoin

- Actor model
  - https://github.com/actix/actix
  - unit test problem
  - service framework on actor model

Actor Service

```rust
pub struct NetworkActorService {
    worker: Option<NetworkWorker>,
    inner: Inner,

    network_worker_handle: Option<AbortHandle>,
}

impl ActorService for NetworkActorService {
    fn started(&mut self, ctx: &mut ServiceContext<Self>) -> Result<()> {}

    fn stopped(&mut self, ctx: &mut ServiceContext<Self>) -> Result<()> {}
}

impl EventHandler<Self, SyncStatusChangeEvent> for NetworkActorService {
    fn handle_event(&mut self, msg: SyncStatusChangeEvent, _ctx: &mut ServiceContext<Self>) {}
}

impl EventHandler<Self, Event> for NetworkActorService {
    fn handle_event(&mut self, event: Event, ctx: &mut ServiceContext<NetworkActorService>) {}
}
```

Async in Starcoin

- future work
  - remote actor
  - rpc on actor

Starcoin

- https://starcoin.org/
- https://github.com/starcoinorg
- We are hiring
- fanngyuan@gmail.com