深入浅出Tokio异步编程

@fanngyuan
Westar实验室

Outline

- Rust network programming
- Futures
- Tokio
- Async/await
- Others

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

Your program

Tokio

Mio

Futures

System selector
(epoll/kqueue()/IOCP/etc.)

Futures

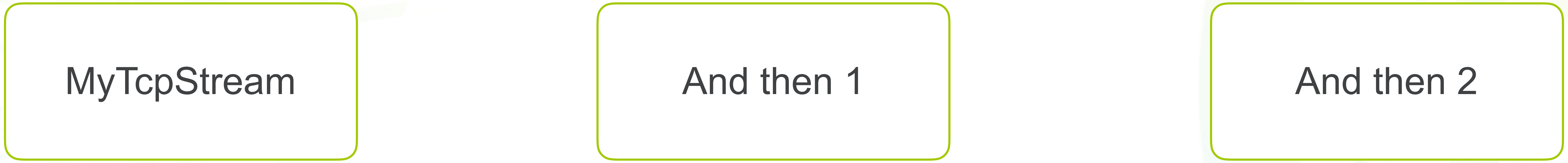
What’s future?

- Database Query
- Rpc
- ……

pull , not push

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

// this is push model
```

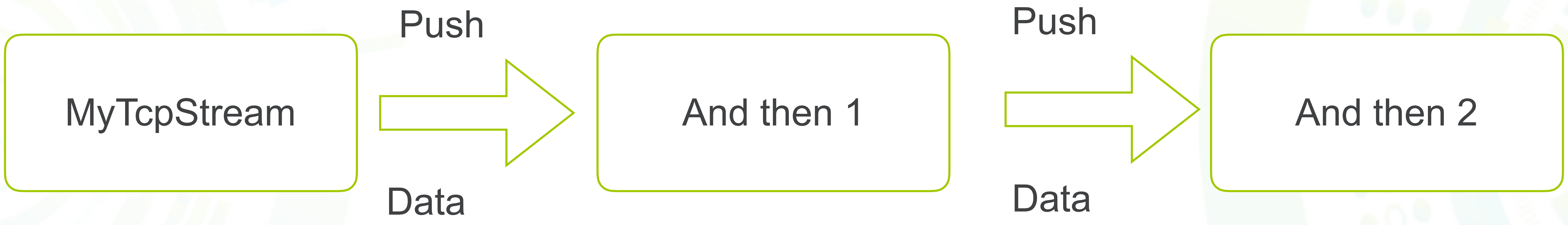
```
let f=MyTcpStream::connect(&remote_addr)
  .and_then(|sock| io::read_exact(sock, 10))
  .and_then(|(sock, response)| {
    process(response)
  });

tokio::spawn(f);
```



MyTcpStream

And then 1

And then 2



MyTcpStream

Push

Data

And then 1

Push

Data

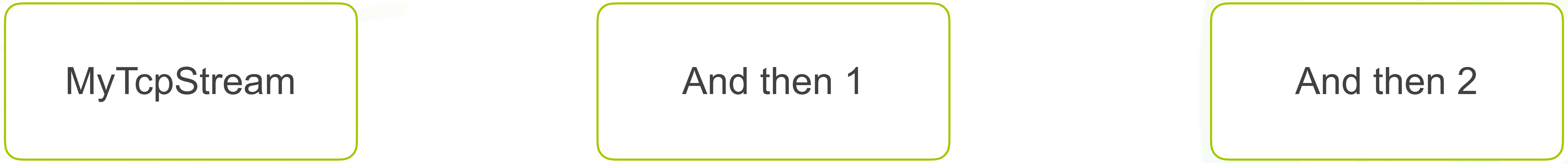
And then 2

Poll future

```
pub trait Future {
    type Item;
    type Error;

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

```
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,
            }
        }
    }

}
```

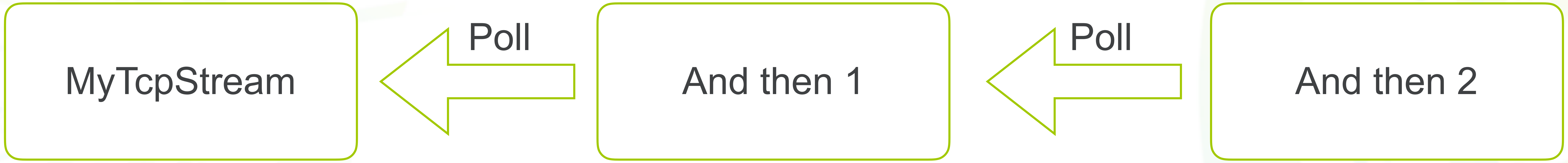
```
enum AndThen<A,F> {
    First(A, F),
}

fn poll(&mut self) -> Async<Item> {
    match fut_a.poll() {
        Async::Ready(v) => f(v),
        Async::NotReady => Async::NotReady,
    }
}
```



MyTcpStream

And then 1

And then 2



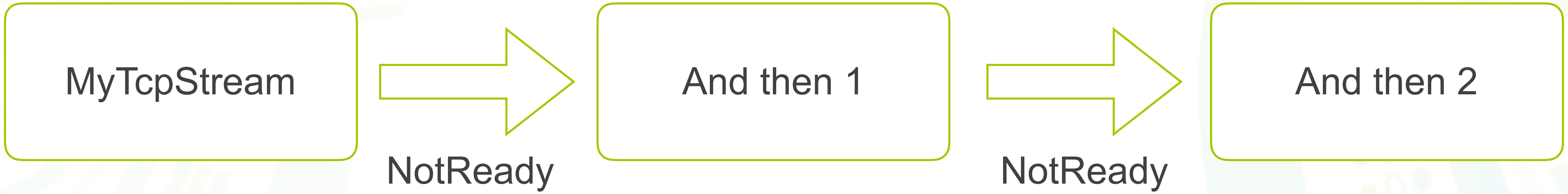
MyTcpStream

Poll

And then 1

Poll

And then 2



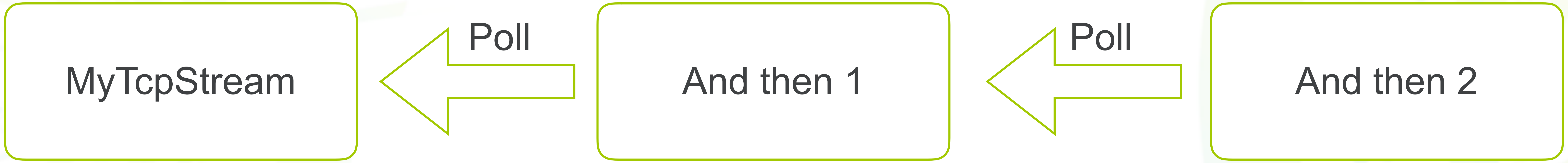
MyTcpStream

NotReady

And then 1

NotReady

And then 2



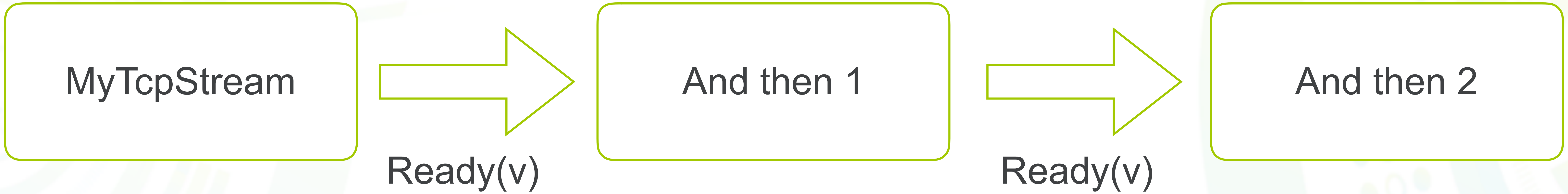
MyTcpStream

Poll

And then 1

Poll

And then 2



MyTcpStream

Ready(v)

And then 1

Ready(v)

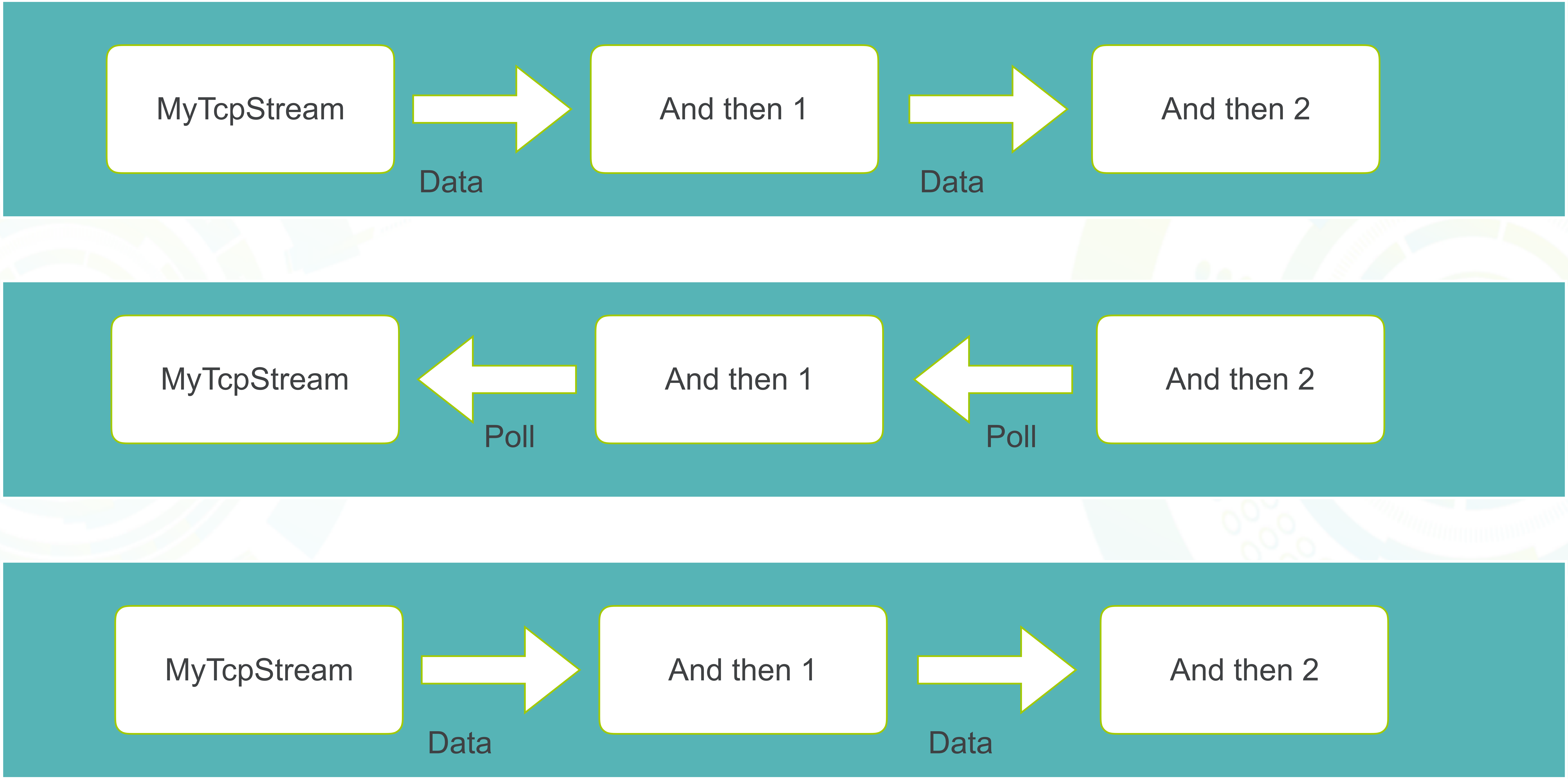
And then 2

- 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

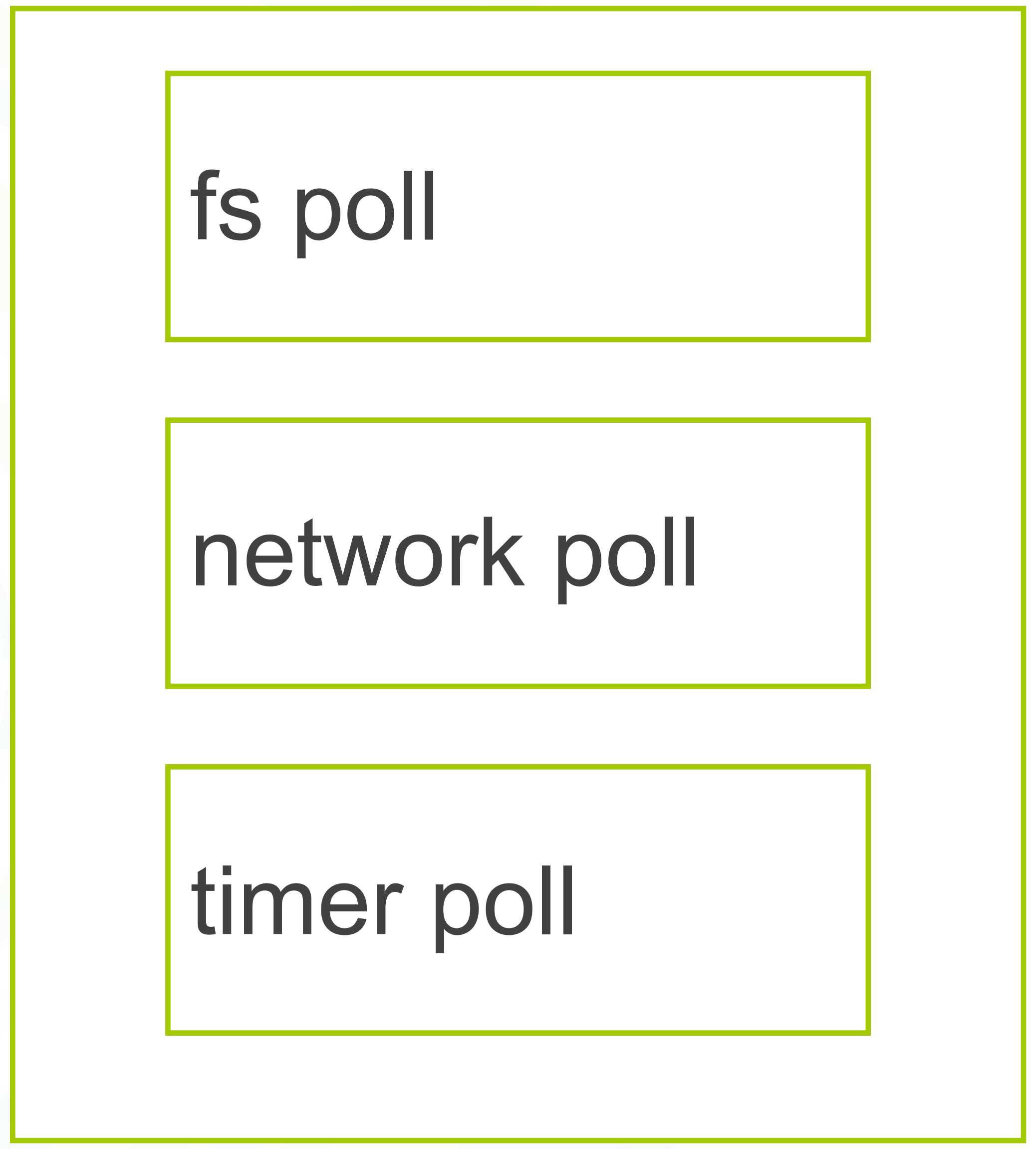
Data

And then 2

Tokio

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

```
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);
```

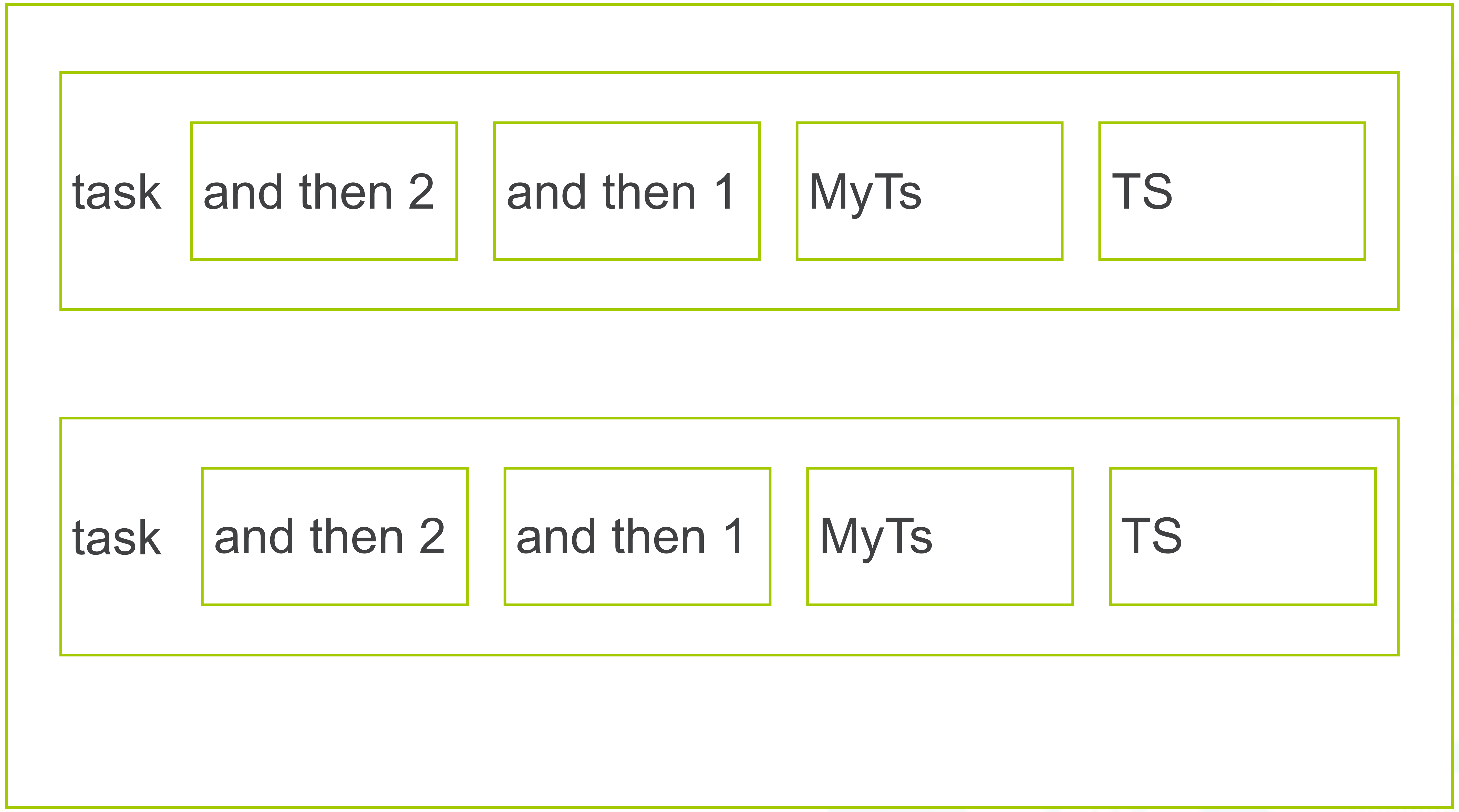


fs poll

network poll

timer poll

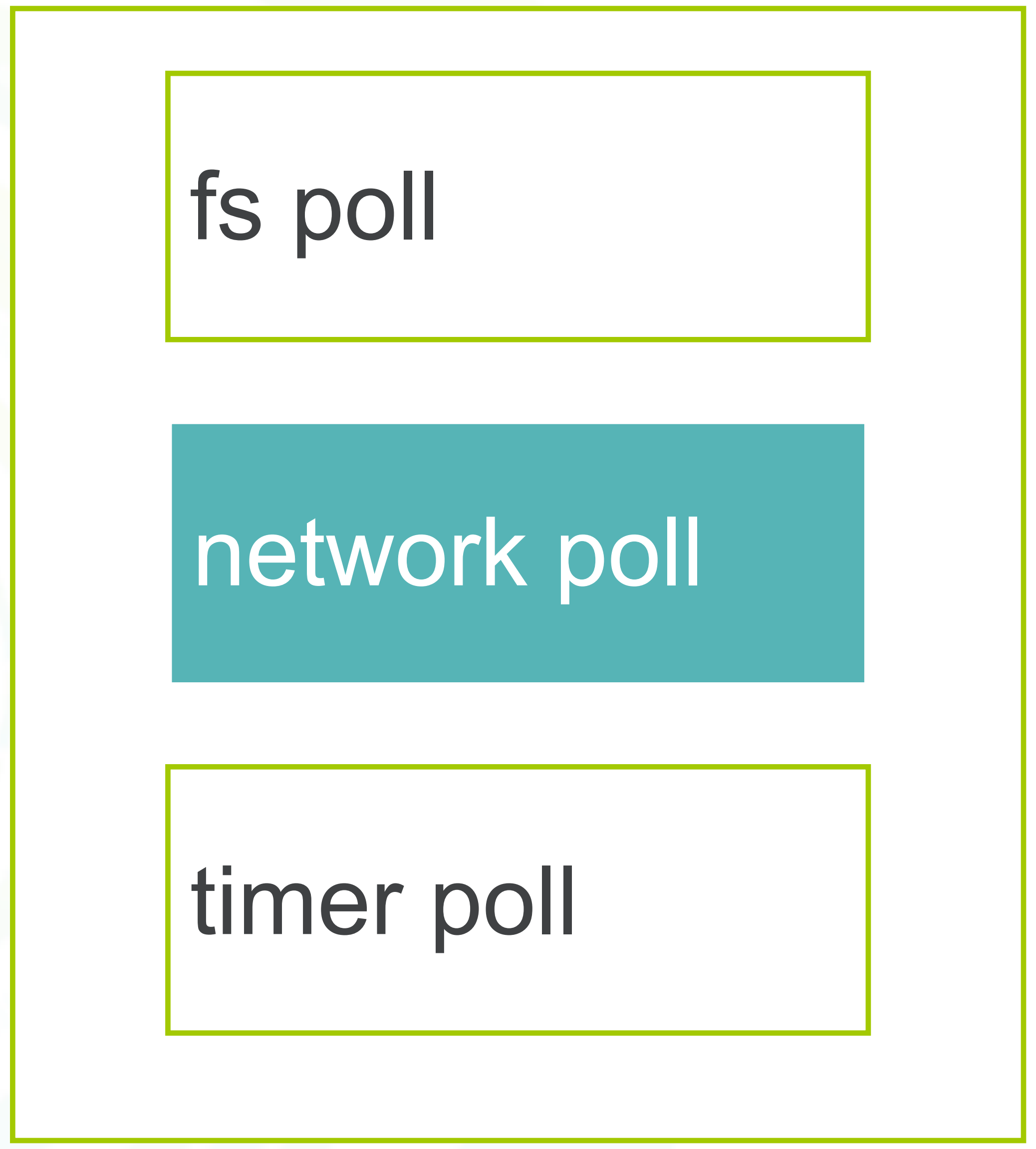
Reactor



task and then 2 and then 1 MyTs TS

task and then 2 and then 1 MyTs TS

Scheduler

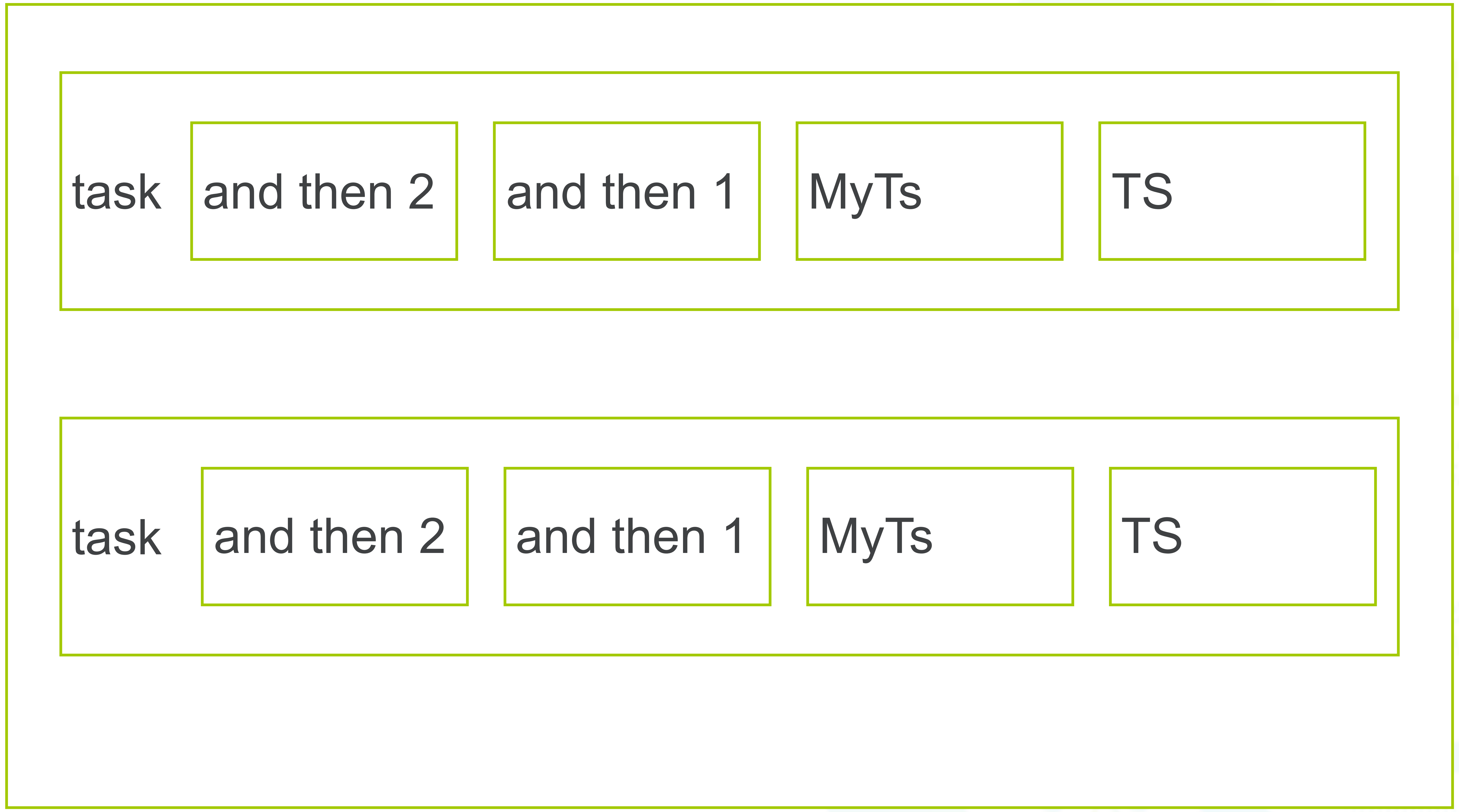


fs poll

network poll

timer poll

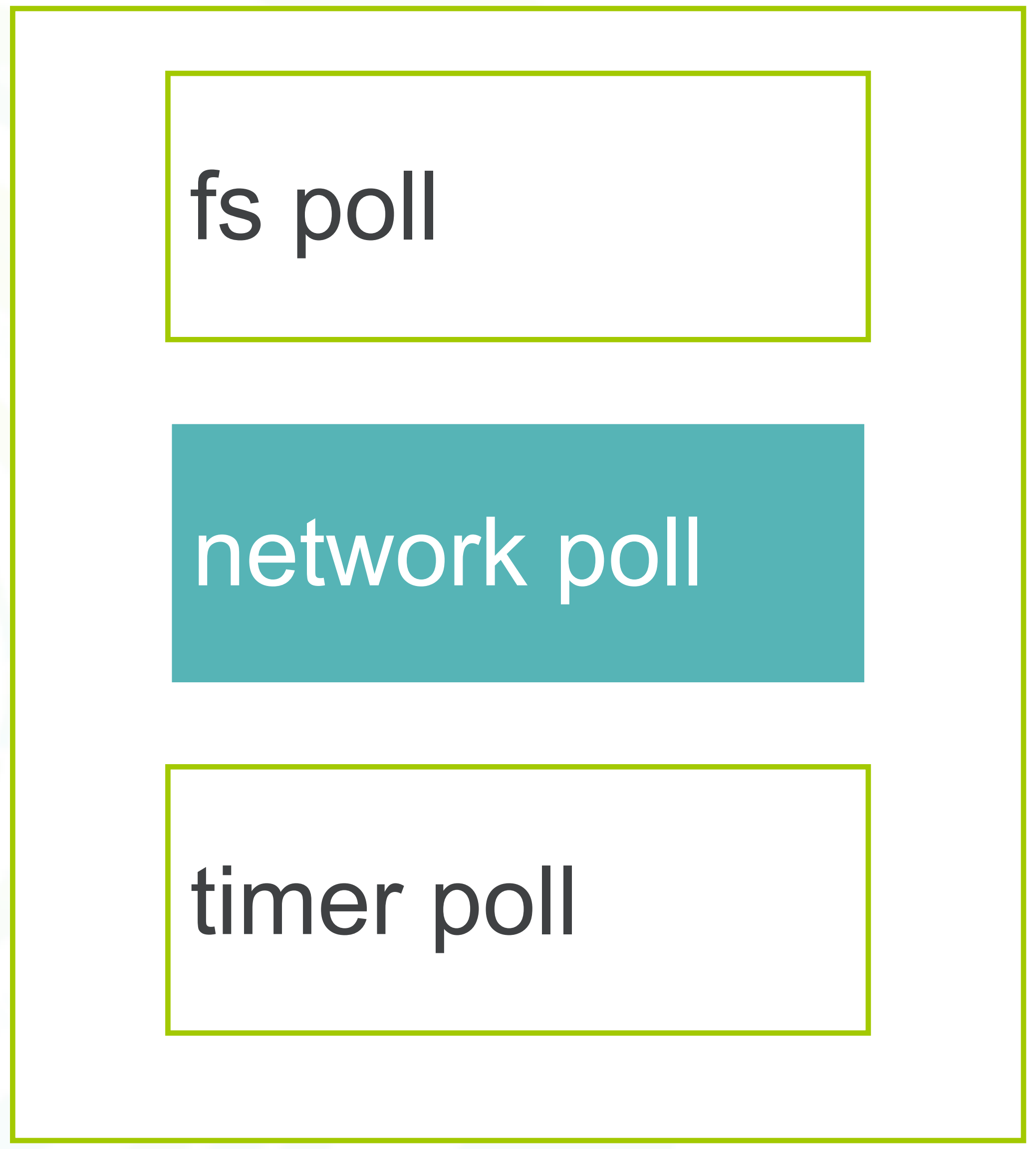
Reactor



task | and then 2 | and then 1 | MyTs | TS

task | and then 2 | and then 1 | MyTs | TS

Scheduler

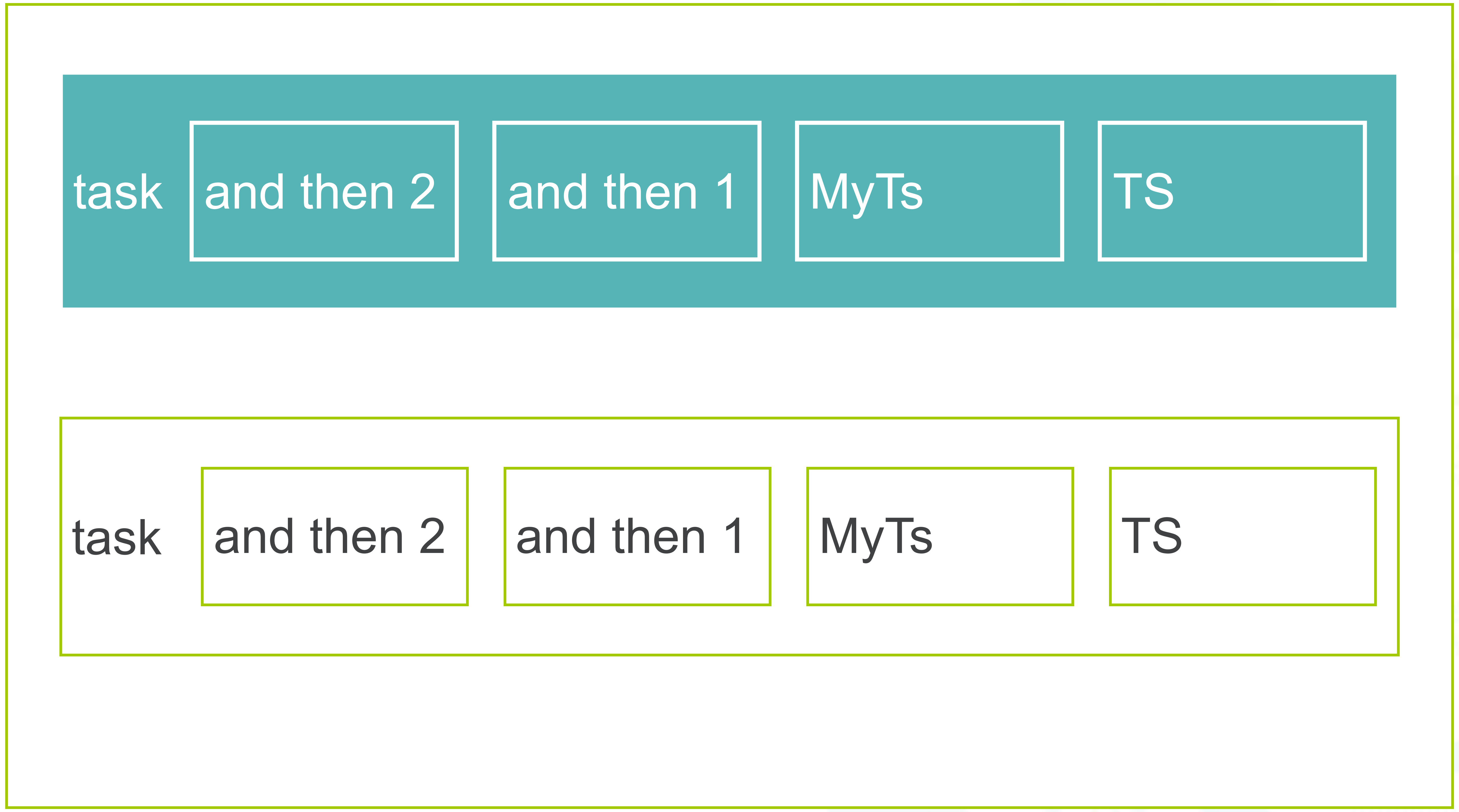


fs poll

network poll

timer poll

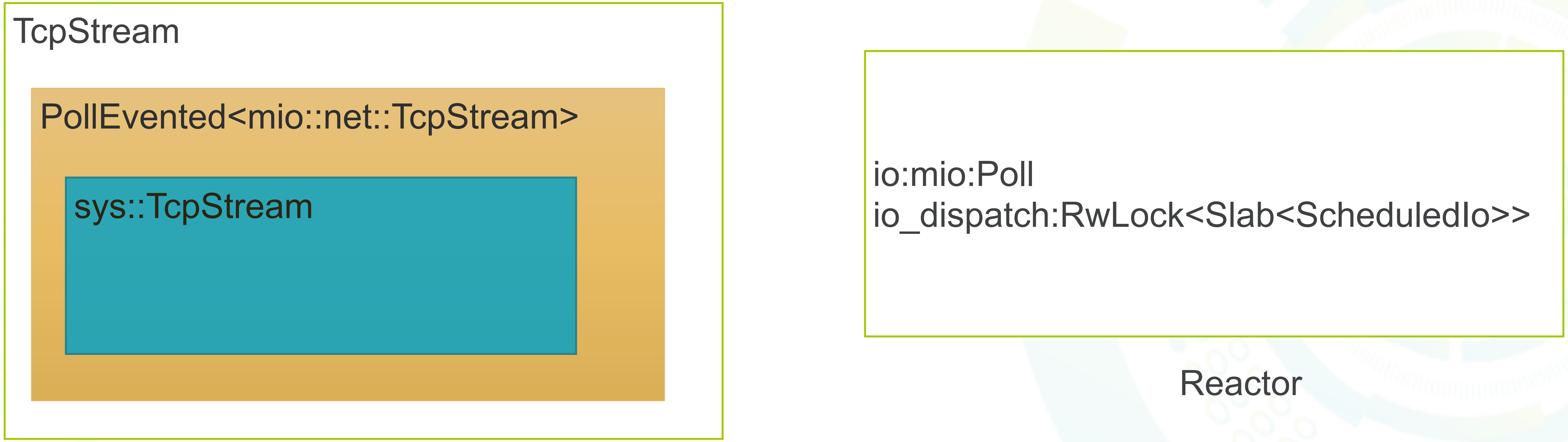
Reactor



task and then 2 and then 1 MyTs TS

task and then 2 and then 1 MyTs TS

Scheduler



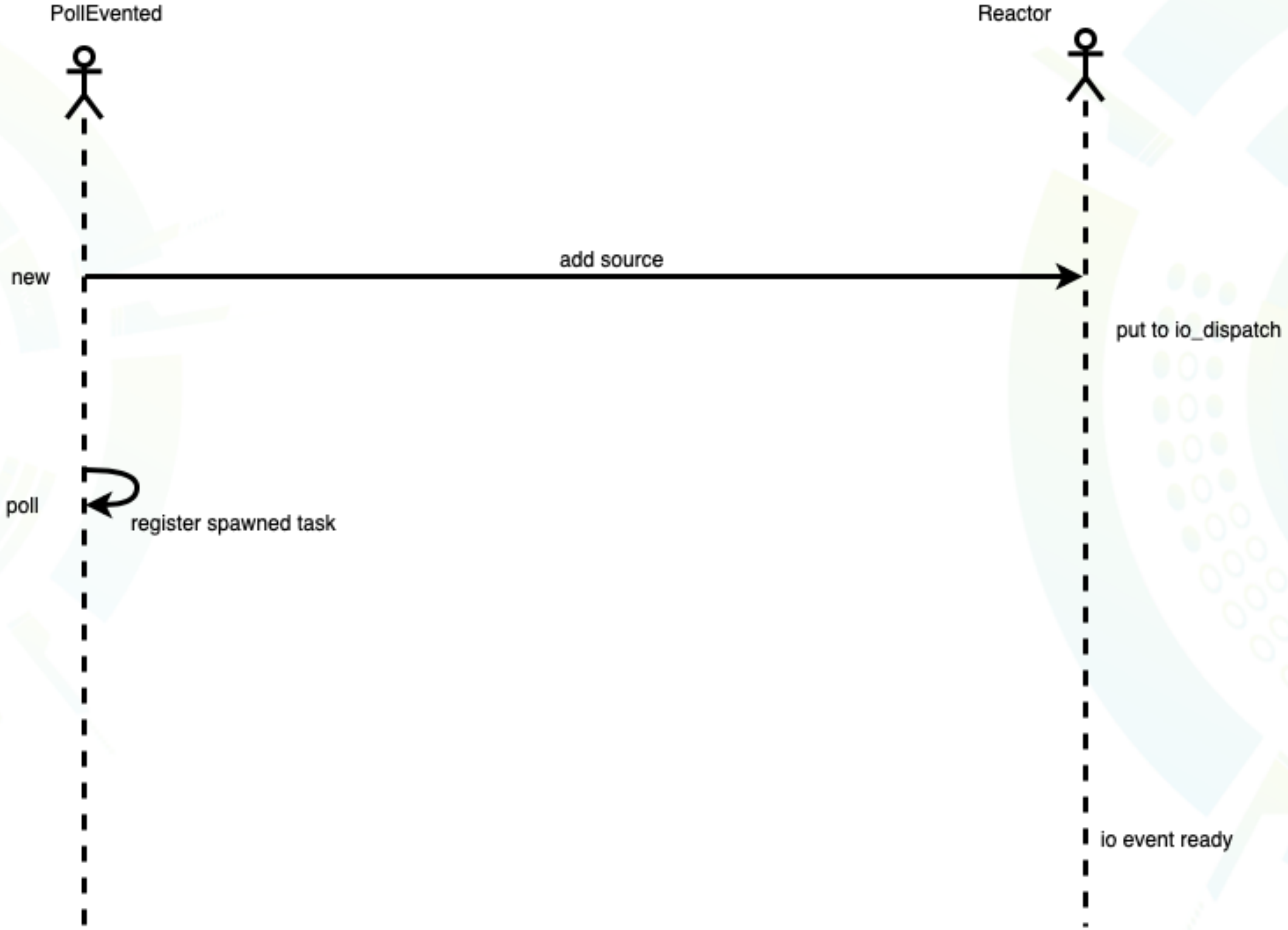
TcpStream

PollEvented<mio::net::TcpStream>

sys::TcpStream

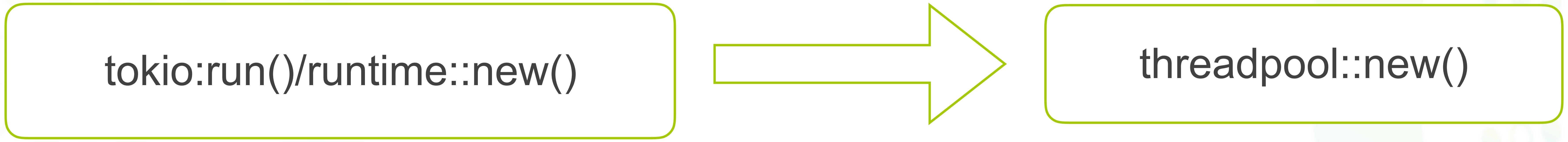
io:mio:Poll
io_dispatch:RwLock<Slab<ScheduledIo>>

Reactor


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

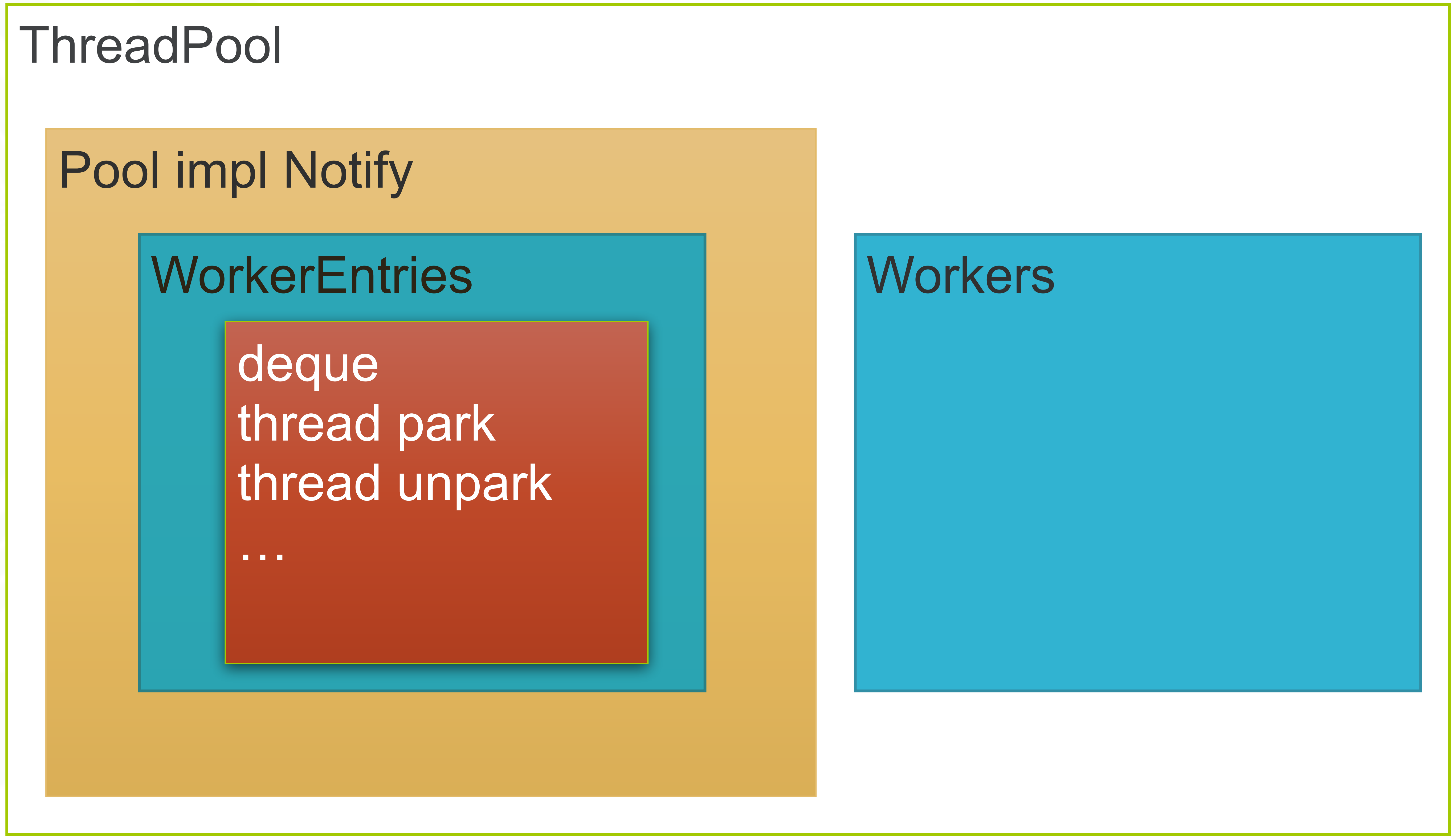
```
task::current() -> Task
Task::notify()
```

```
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()



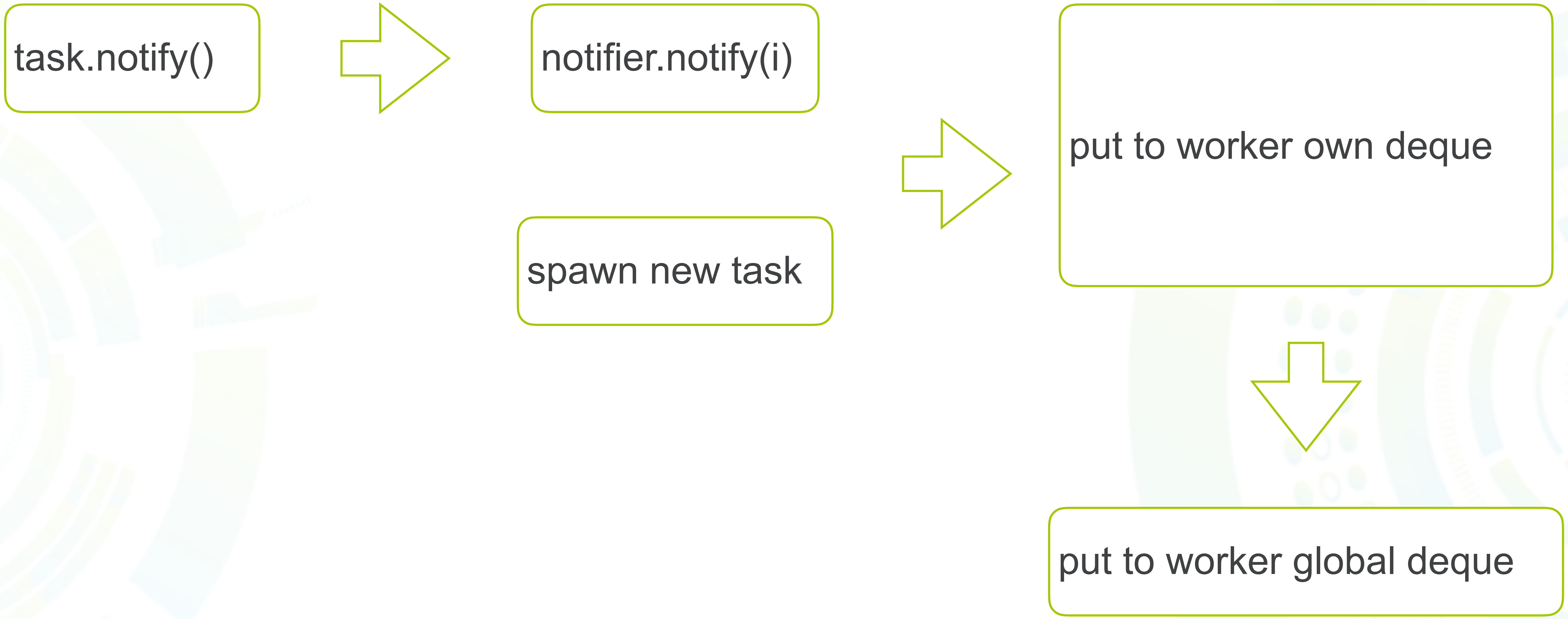
ThreadPool

Pool impl Notify

WorkerEntries

deque
thread park
thread unpark
…

Workers



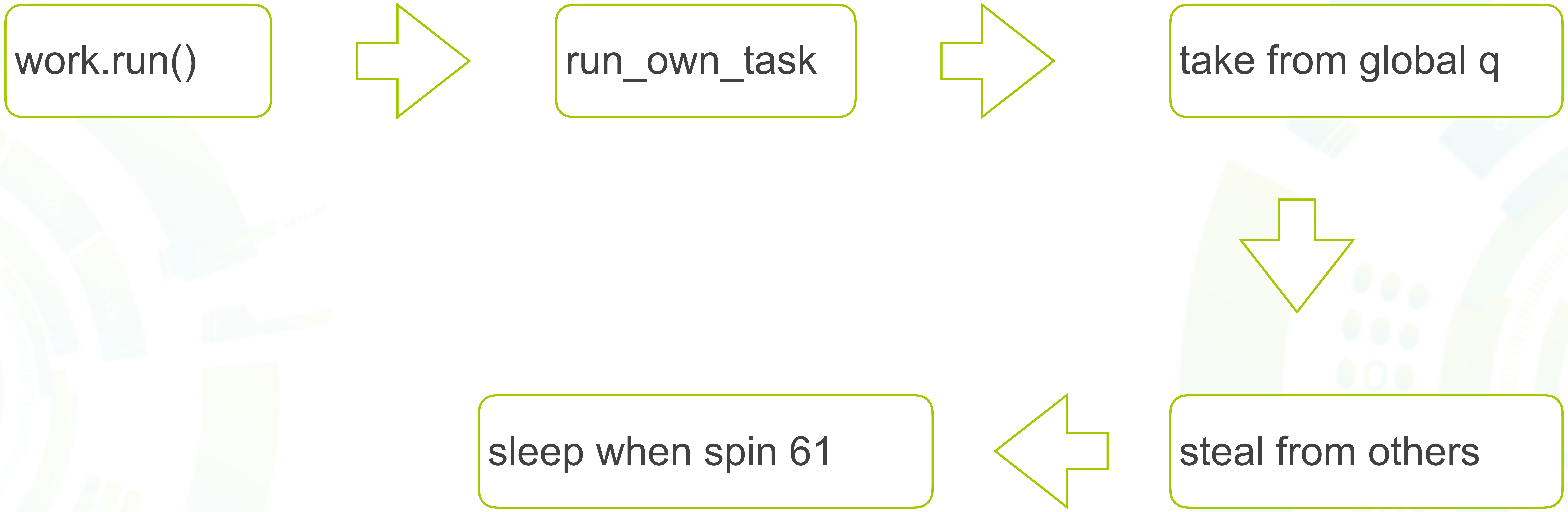
task.notify()

notifier.notify(i)

spawn new task

put to worker own deque

put to worker global deque



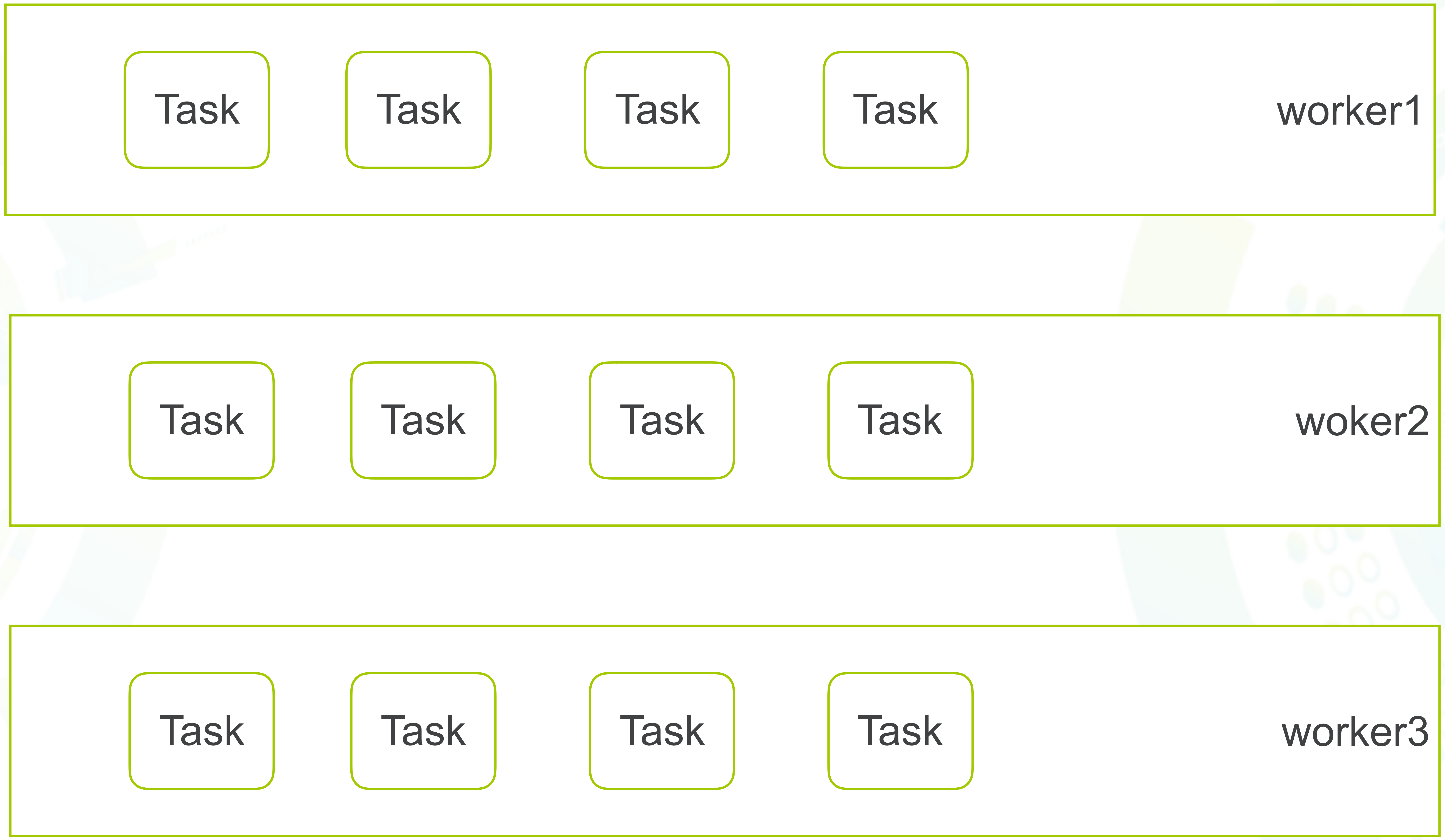
work.run()

run_own_task

take from global q

steal from others

sleep when spin 61



Task Task Task Task worker1

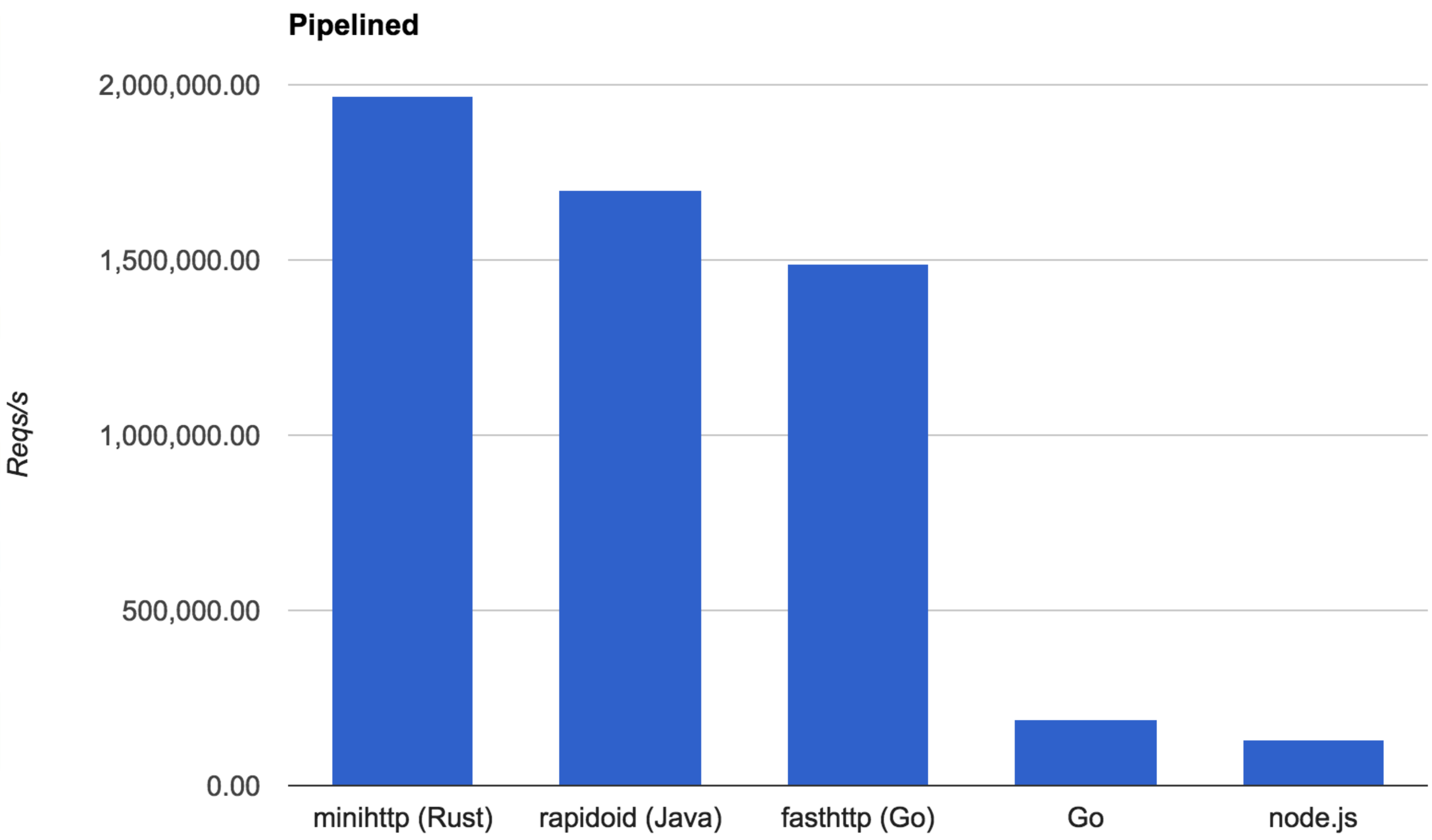
Task Task Task Task woker2

Task Task Task Task worker3

async/await

```
#[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(())
}
```

Others



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/await

Best Practice

- Runtime
  - Different runtime for different business
- TaskExecuto
  - tokio::spawn for default runtime
  - get from runtime
  - clone
- Actor model
- async/await