Using asynchronous I/O in Rust

(was: Techniques for writing concurrent applications with asynchronous I/O)

Matthieu Wipliez

Upstream Studies Engineer

September, 17th, 2016

About: you, this talk, and the speaker

- About you: background and experience with asynchronous I/O?
- This talk
  - Asynchronous I/O for networking
  - Rust code examples accessible on GitHub
- Author of edge-rs: Web framework in Rust
  - Based on Hyper

Synchronous (blocking) I/O

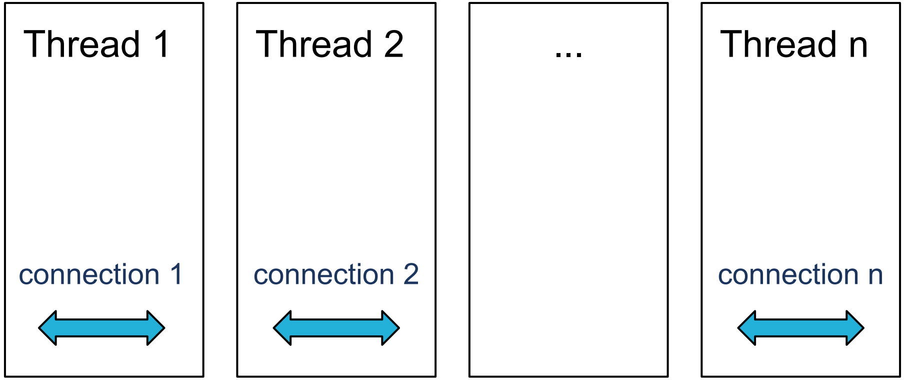
- In Rust, two traits:
  - Read
  - Write

- Typical use:

```
let mut buf = [0; 4096];
let num_bytes = try!(stream.read(&mut buf));
// ...
let mut response = Vec::new();
// ...
stream.write_all(&response)
```

Synchronous I/O architecture

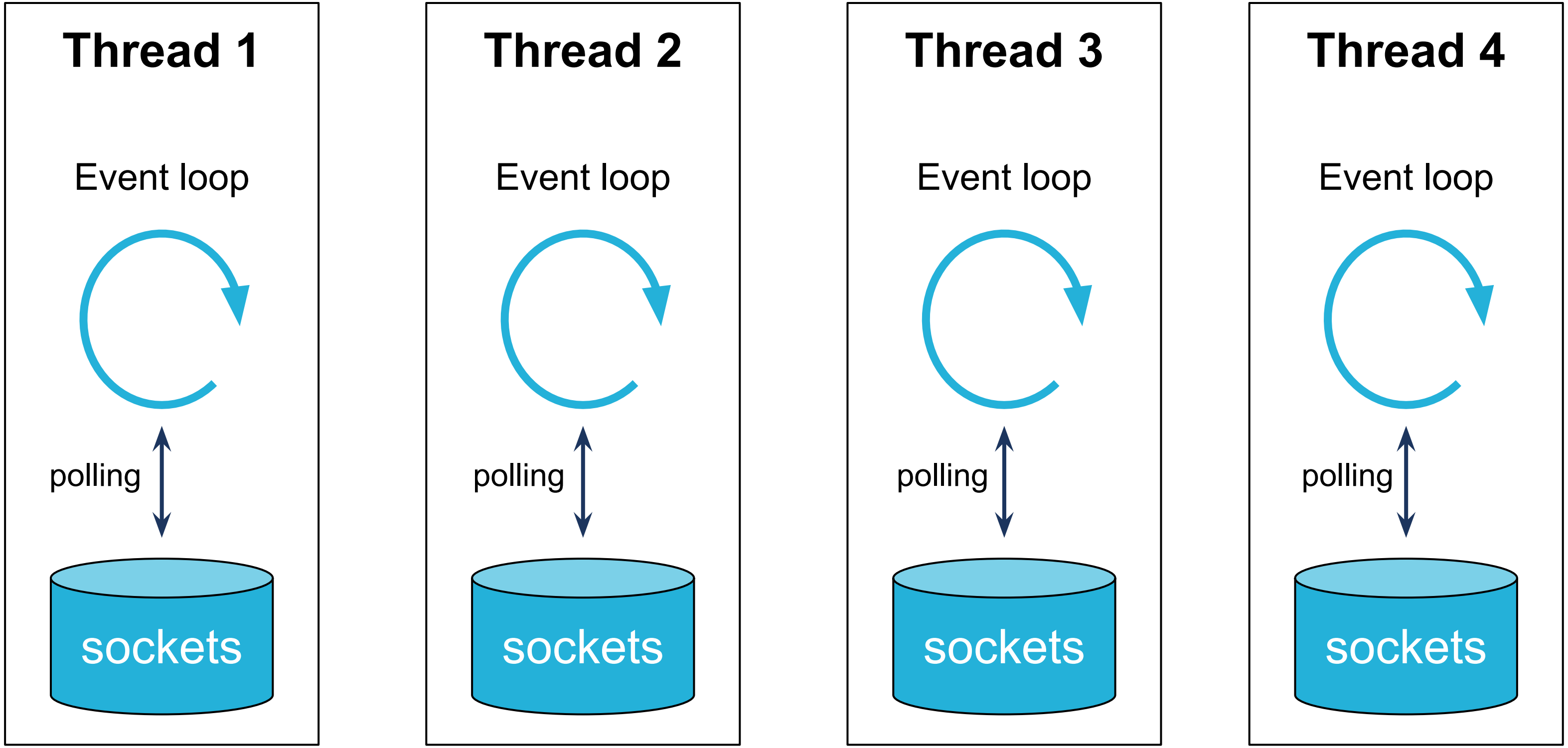
- One or more processes listen on a port
- Spawn one thread per connection



Implementation of a synchronous server in Rust

```rust
let listener = TcpListener::bind("0.0.0.0:113").unwrap();

for stream in listener.incoming() {
    let stream = stream.unwrap();
    thread::spawn(move || {
        // connection handler
        handle_connection(stream)
    });
}
```

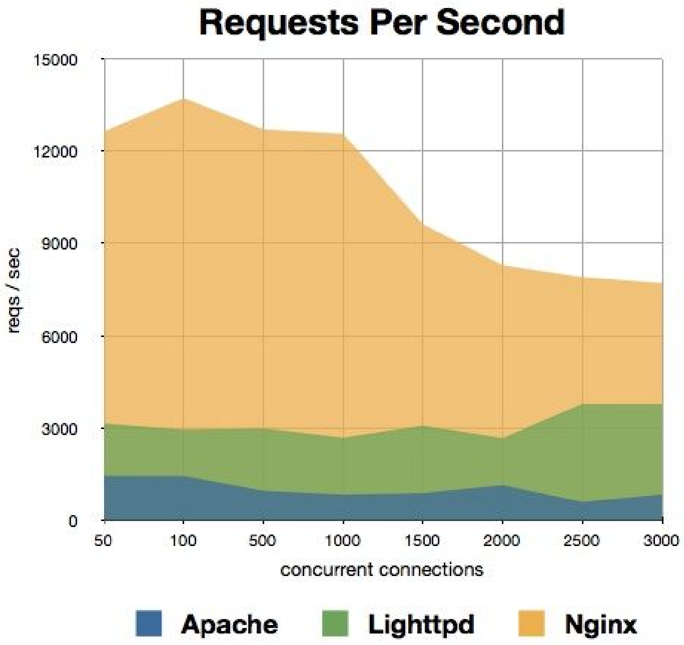
Asynchronous I/O architecture

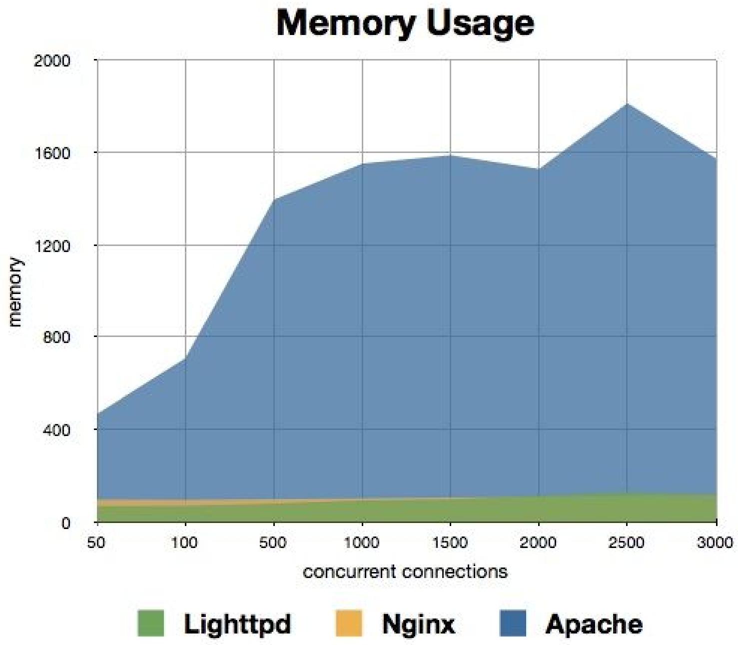


Advantages of asynchronous I/O

- Throughput: number of requests per second
- Latency: time to serve a request
- Memory consumption

Asynchronous I/O in Web servers

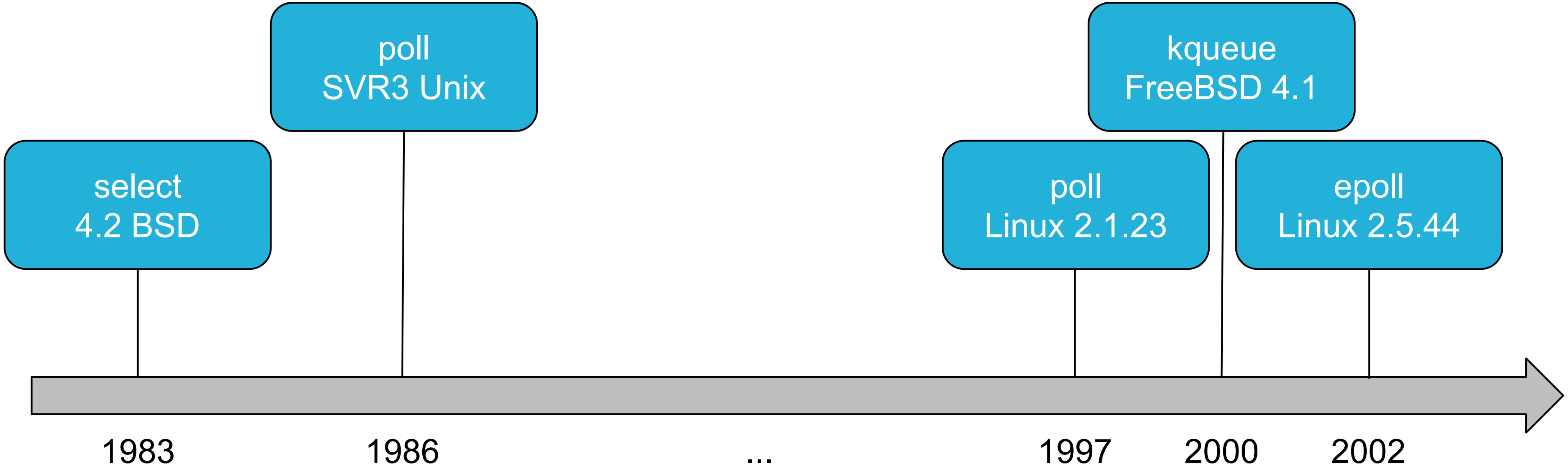




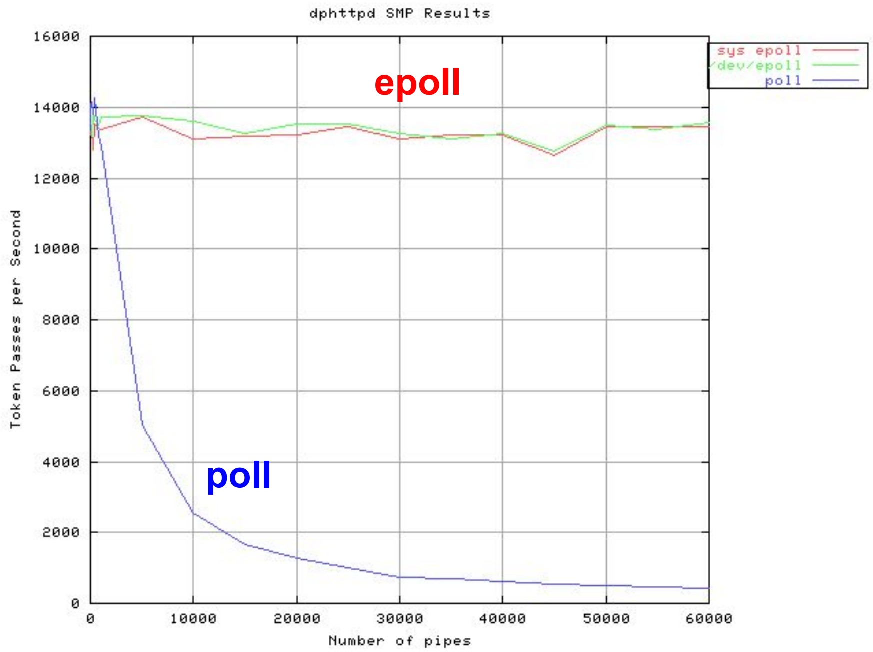
© 2016 DreamHost

30+ years of asynchronous I/O in Unix

“The classic Unix way to wait for I/O events on multiple file descriptors is with the select() and poll() system calls.” [Jonathan Corbet http://lwn.net/Articles/14168/]



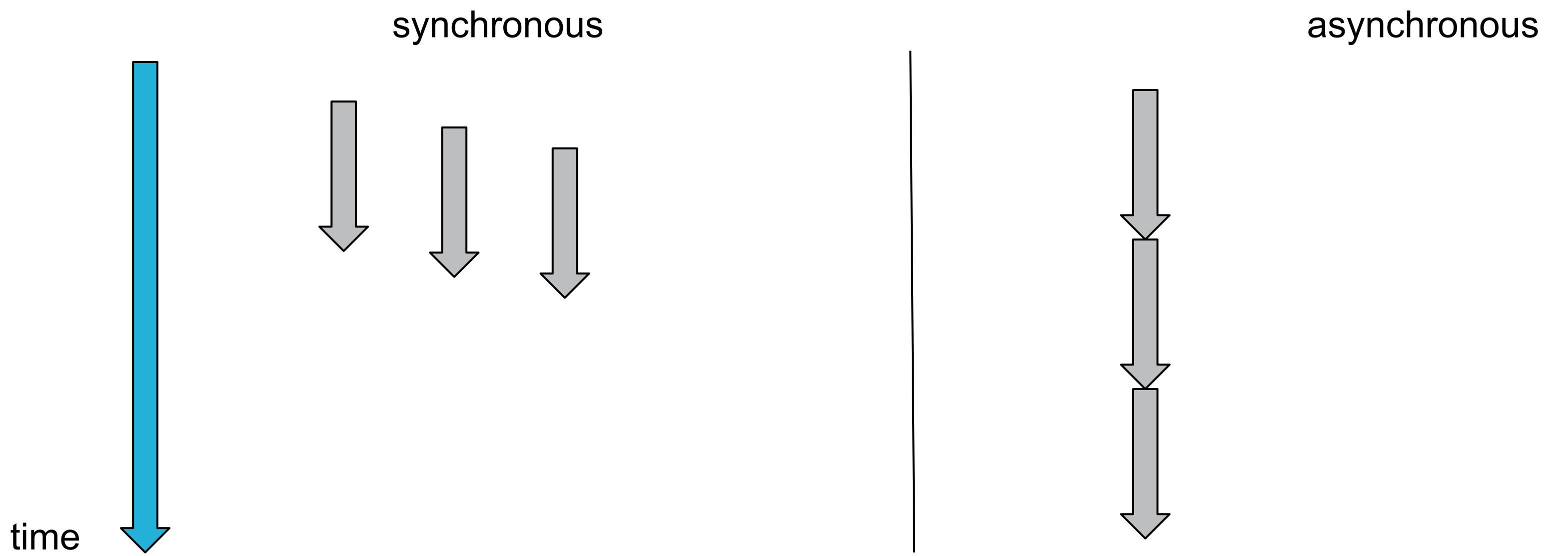
Side note: implementing polling for scalability



© 2002 Davide Libenzi

Something to keep in mind with asynchronous I/O

- Do *not* block the current thread!
- Otherwise, here is what happens:



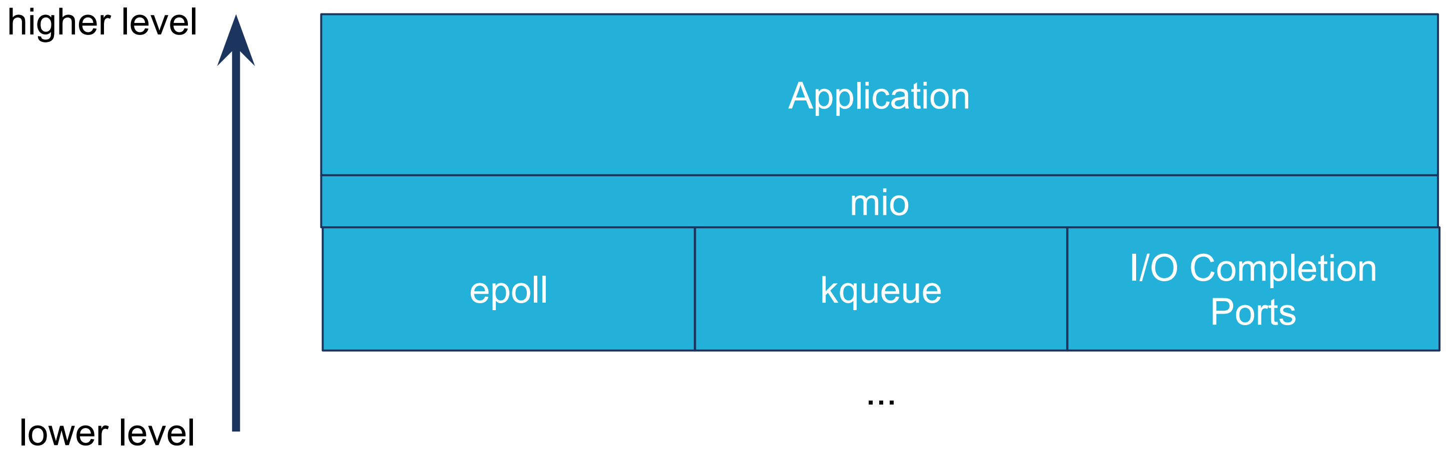
Asynchronous I/O in Rust

Non-blocking I/O in Rust standard library

- API support
  - For structures `TcpListener`, `TcpStream`, `UdpSocket`…
  - Function `set_non_blocking` (since Rust 1.9)
- Semantics
  - When a read or a write “needs to block to complete, but the blocking operation was requested to not occur”, returns `ErrorKind::WouldBlock`
- Limitations: no polling API

mio: Support for asynchronous polling

- Thin wrapper around underlying libraries
- Low-level, zero allocation
- Callbacks with tokens



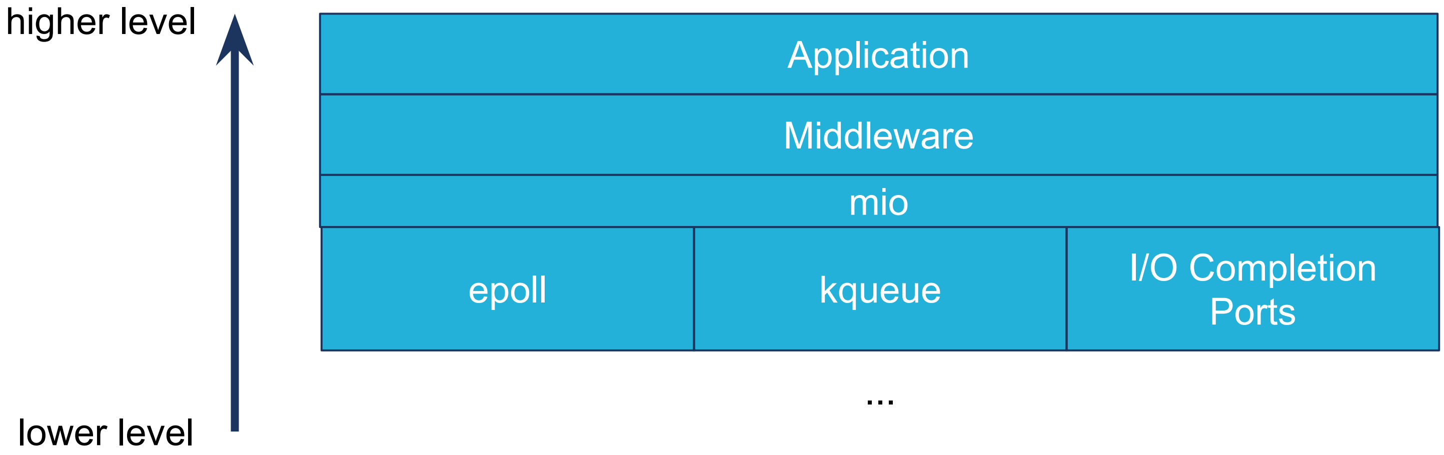
Using mio directly?

```rust
const SERVER: Token = Token(10_000_000);
const CLIENT: Token = Token(10_000_001);
struct EchoConn {
   sock: TcpStream,
   buf: Option<ByteBuf>,
   mut_buf: Option<MutByteBuf>,
   token: Option<Token>,
   interest: Ready
}
type Slab<T> = slab::Slab<T, Token>;
impl EchoConn {
   fn new(sock: TcpStream) -> EchoConn {
       EchoConn {
           sock: sock,
           buf: None,
           mut_buf: Some(ByteBuf::mut_with_capacity(2048)),
           token: None,
           interest: Ready::hup()
       }
   }

  fn readable(&mut self, event_loop: &mut EventLoop<Echo>) ->
    io::Result<()> {
    …
  }
```

```rust
fn writable(&mut self, event_loop: &mut EventLoop<Echo>) -> io::Result<()> {
    let mut buf = self.buf.take().unwrap();
    match self.sock.try_write_buf(&mut buf) {
        Ok(None) => {
            debug!("client flushing buf; WOULDBLOCK");
            self.buf = Some(buf);
            self.interest.insert(Ready::writable());
        }
        Ok(Some(r)) => {
            debug!("CONN : we wrote {} bytes!", r);
            self.mut_buf = Some(buf.flip());
            self.interest.insert(Ready::readable());
            self.interest.remove(Ready::writable());
        }
        Err(e) => debug!("not implemented; client err={:?}", e),
    }

    assert!(self.interest.is_readable() || self.interest.is_writable(), "actual={:?}", self.interest);
    event_loop.reregister(&self.sock, self.token.unwrap(), self.interest,
                          PollOpt::edge() | PollOpt::oneshot())
}
…
}
```

Using mio in practice

- If possible, use a higher-level library on top of mio
  - Such as: rotor, mioco, coio, **tokio**



Why Tokio?

- Distinctive features:
  - No need to register interest
  - Futures-based asynchronous I/O

- Futures (a.k.a promise)
  - Deferred computation
  - Solves “callback hell”

Before Tokio: callbacks and interest

For instance Handler trait in Hyper:

```
pub trait Handler<T: Transport> {
    fn on_request(&mut self, request: Request<T>) -> Next;
    fn on_request_readable(&mut self, request: &mut http::Decoder<T>) -> Next;
    fn on_response(&mut self, response: &mut Response) -> Next;
    fn on_response_writable(&mut self, response: &mut http::Encoder<T>) -> Next;
}
```

From callbacks to futures

Before

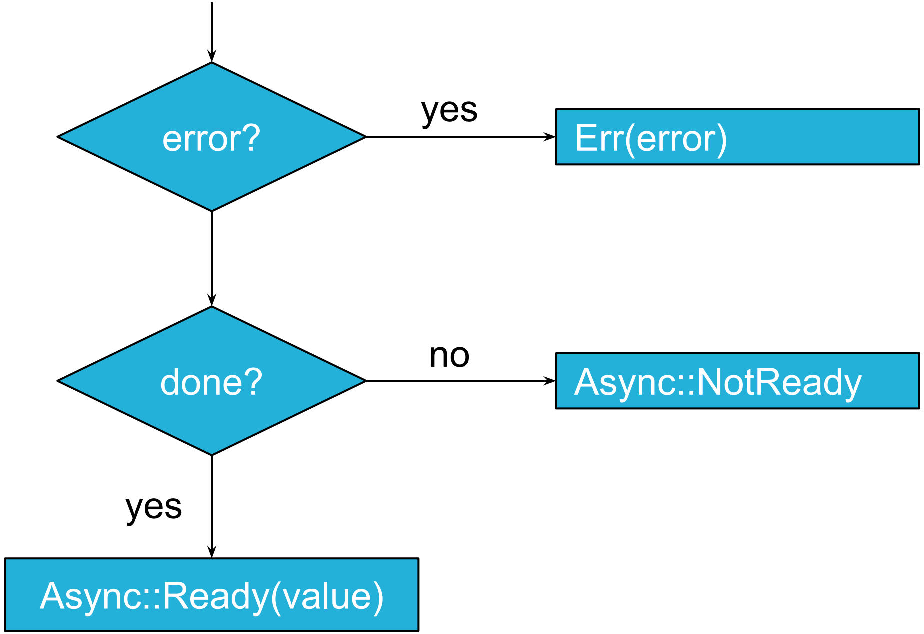
```
if let Ready(addr) = resolve(url) {
    if let Ready(tcp) = connect(&addr) {
        if let Ready(data) = download(tcp) {
            // ...
        }
    }
}
```

After

```
let addr = resolve(url);
let tcp = addr.and_then(|addr| connect(&addr));
let data = tcp.and_then(|conn| download(conn));
// ...
```

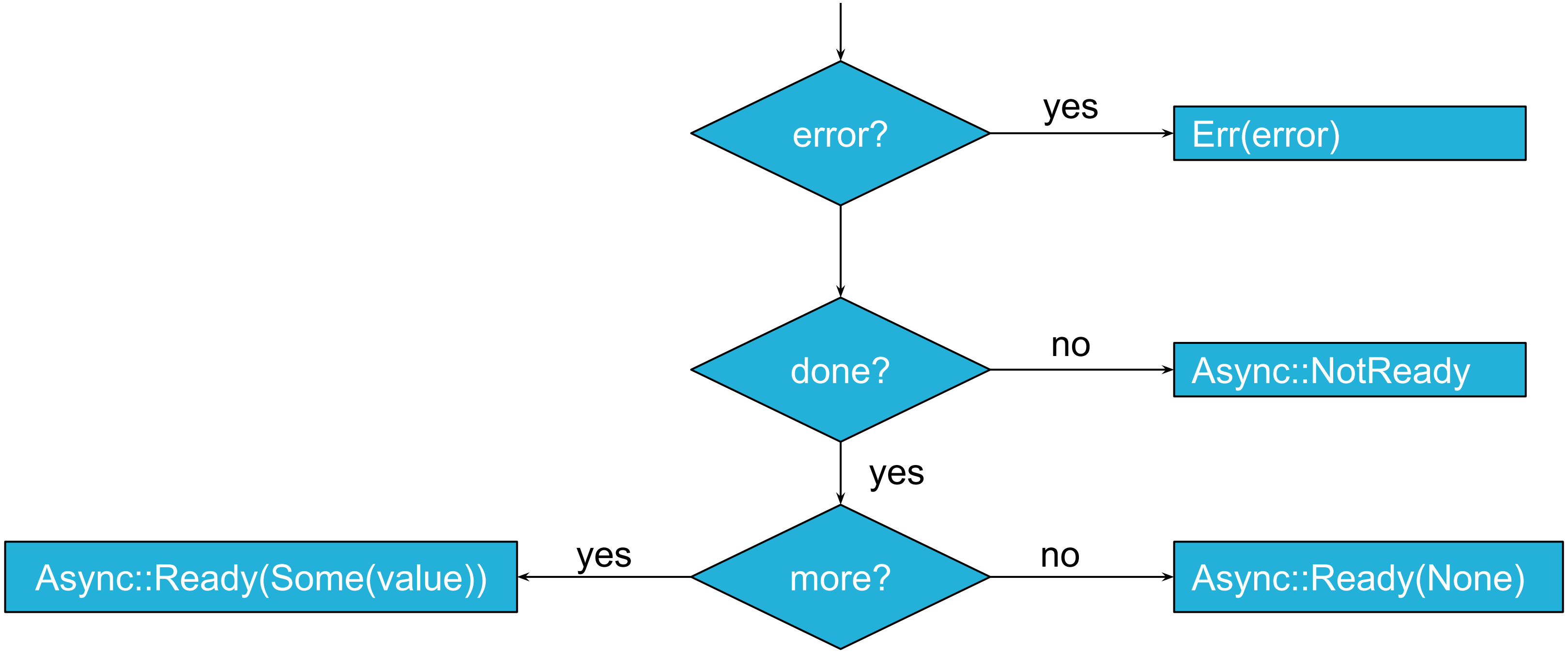
Definition of Future

```
fn poll(&mut self) -> Result<Async<T>, E>;
```



Non-blocking iterator: Stream

```
fn poll(&mut self) -> Result<Async<Option<T>>, E>;
```



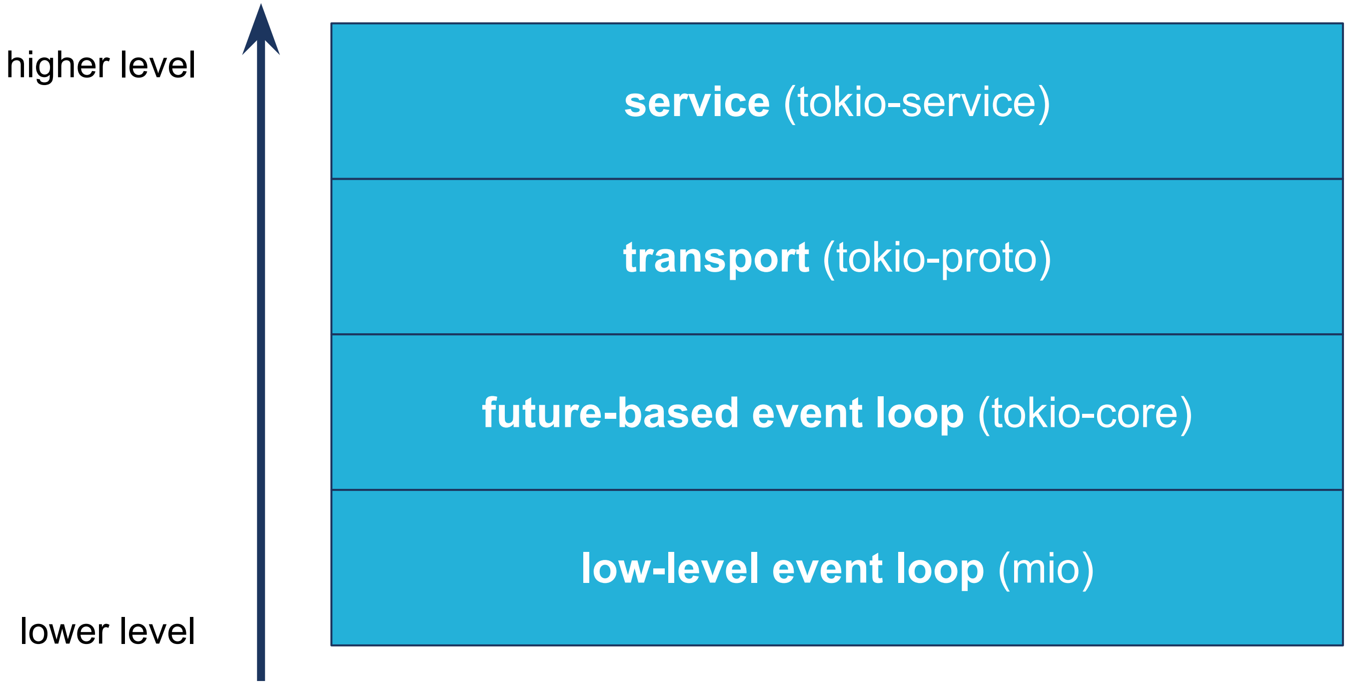
Futures and (not) blocking

- Do not block a future. Never *wait* in a future!

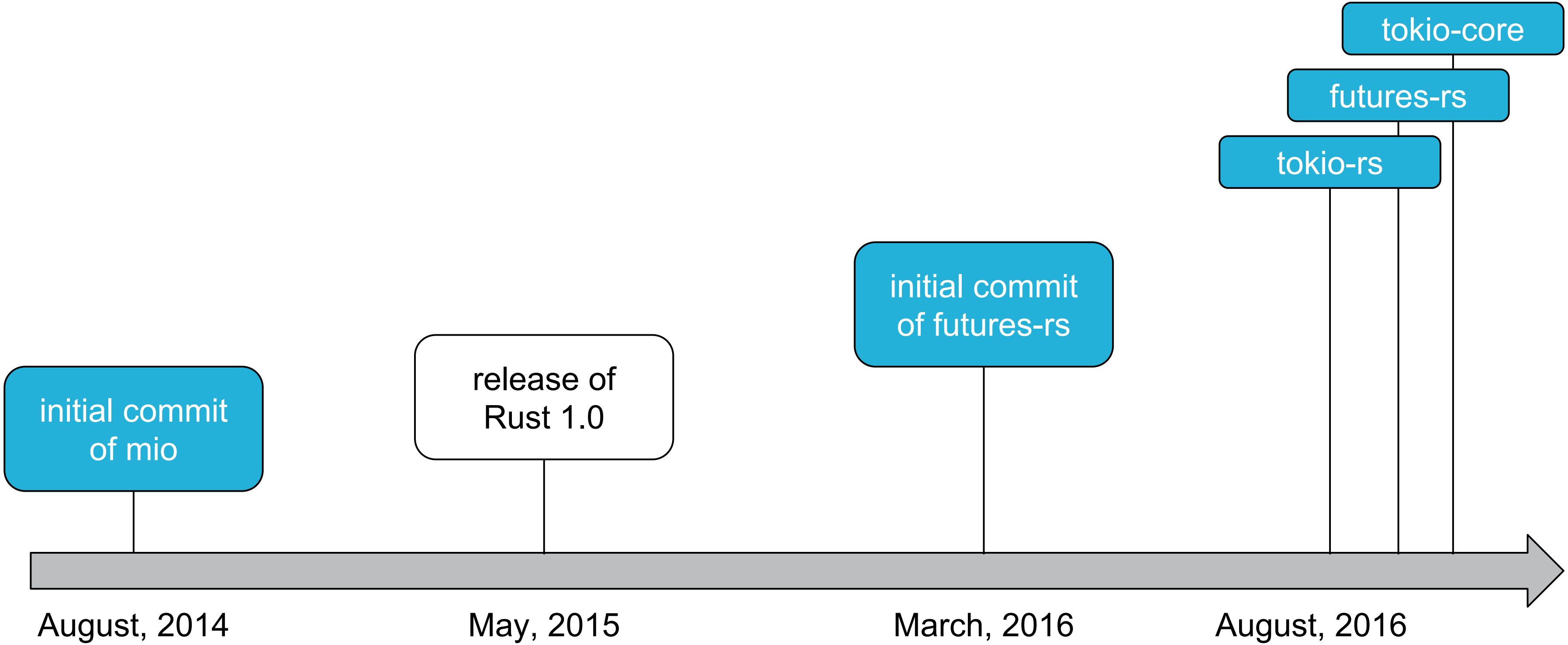
- What if your program needs to:
  - Do compute-intensive work?
  - Call blocking functions?

- Then: use a thread pool
  - A number of worker threads executing jobs
  - For futures: futures_cpupool

Tokio stack

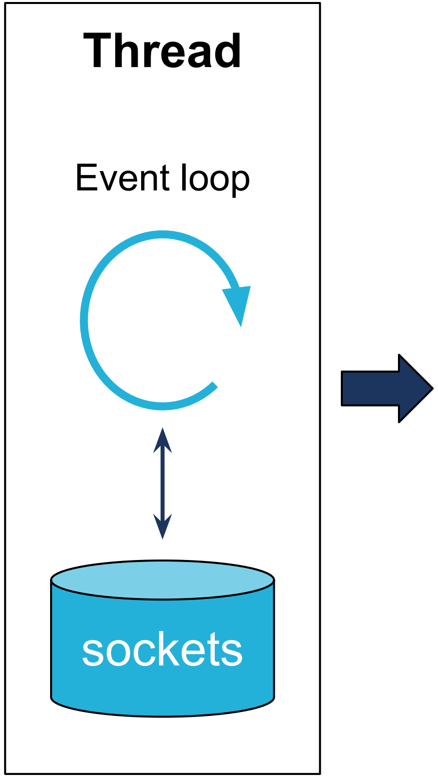

higher level
lower level
service (tokio-service)
transport (tokio-proto)
future-based event loop (tokio-core)
low-level event loop (mio)

History of Tokio and futures



Event loop in Tokio

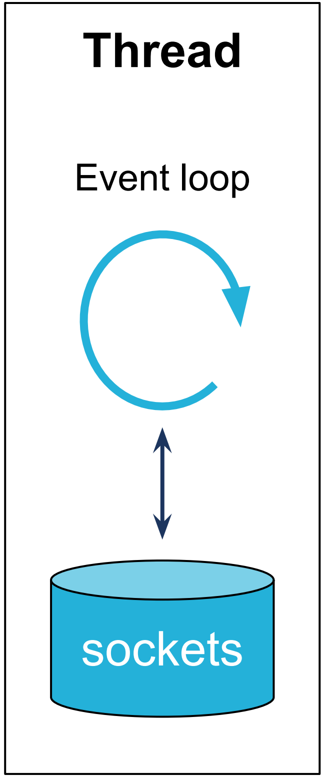
harmonic




```rust
use tokio_core::reactor::Core;

let mut event_loop = Core::new().unwrap();

// let listener = ...

let future = listener.incoming().for_each(|_| {
    // ...
});

event_loop.run(future).unwrap();
```

Handling connections

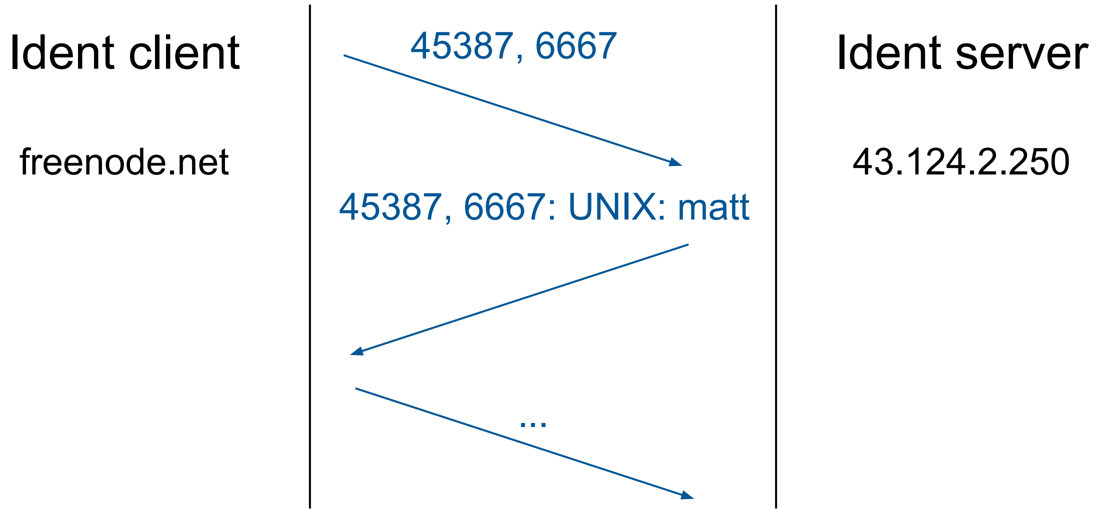
harmonic




```rust
use tokio_core::reactor::Core;

let mut event_loop = Core::new().unwrap();
let handle = event_loop.handle();

// ...

let future = listener.incoming().for_each(|(stream, _)| {
    // ...
    handle.spawn(MyHandler::new(stream));
    Ok(())
});

event_loop.run(future).unwrap();
```

Use case: Identification Protocol (RFC 1413)

- Goal: identify the user of a particular TCP connection
- Example: who is connected on freenode.net on IRC from 43.124.2.250:45387 ?



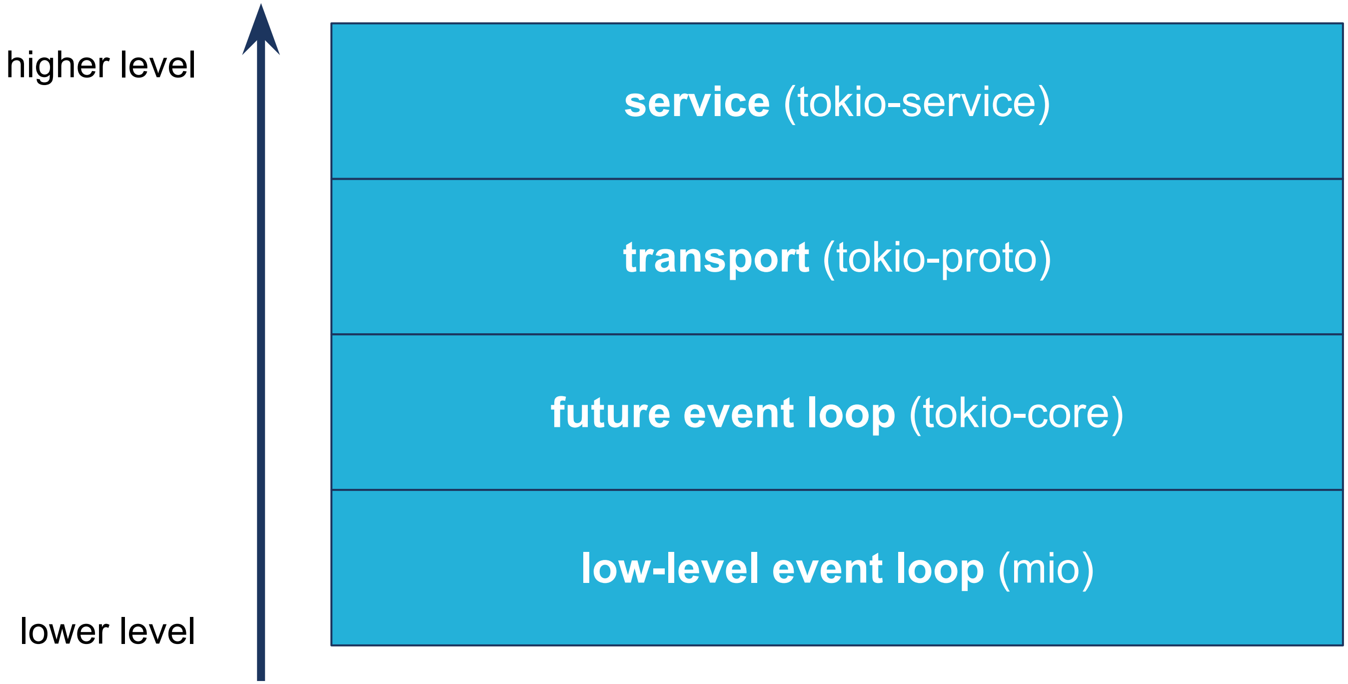
First solution using tokio-core

```
impl Future for IdentHandler {
    type Item = ();
    type Error = io::Error;

    fn poll(&mut self) -> Poll<(), io::Error> {
        if try!(self.stream.read_line(&mut self.request)) > 0 {
            let reply = self.handle();
            try!(self.stream.get_ref().write_all(reply.as_ref()));

            self.request.clear();
            return Ok(Async::NotReady);
        }

        Ok(Async::Ready(())) // EOF
    }
}
```

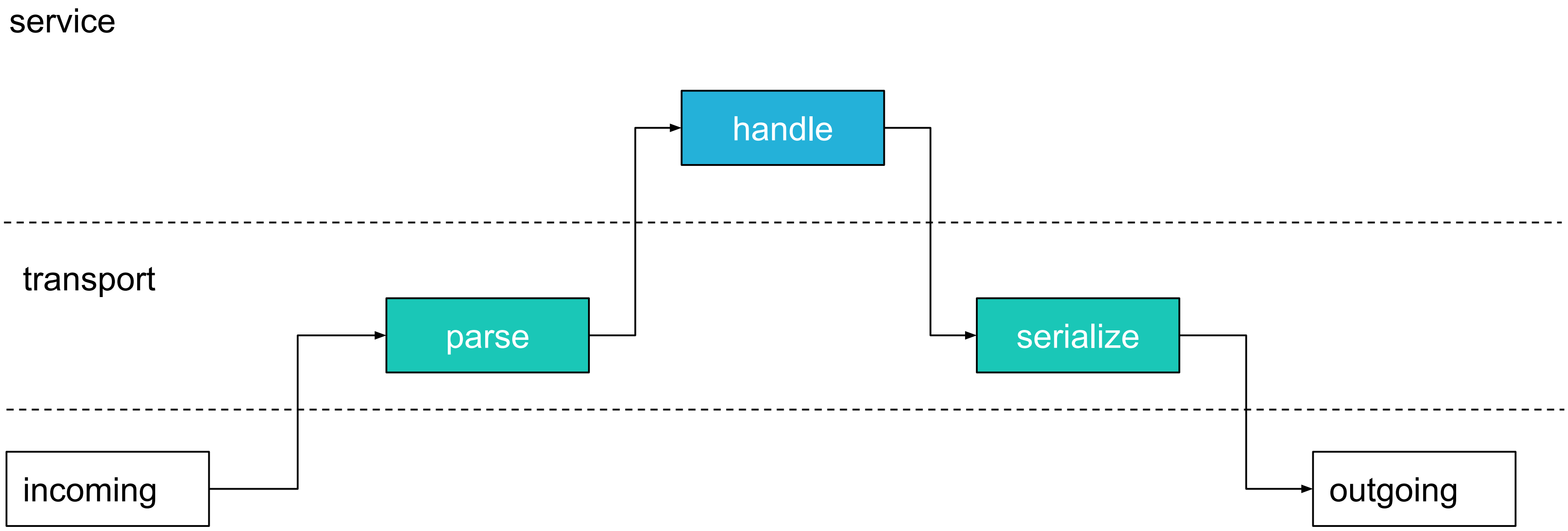
Fixing our implementation

- Calling read_line would block
  - BufRead::read_line -> BufRead::read_until -> BufRead::fill_buf -> Read::read
  - `const DEFAULT_BUF_SIZE: usize = 8 * 1024;`
- Calling write_all could block
  - `write` would not block, but it may only write a subset of data
- Possible solutions: intermediate buffers, streams

Another solution using Tokio service



Request handling with Tokio service architecture



Implement the transport

```
impl Parse for MyParser {
    type Out = Request;

    fn parse(&mut self, buf: &mut BlockBuf) -> Option<Request> {
        // ...
    }
}
```

```
impl Serialize for MySerializer {
    type In = Response;

    fn serialize(&mut self, frame: Response, buf: &mut BlockBuf) {
        // ...
    }
}
```

Putting it all together (simplified)

```rust
use tokio::service::simple_service;

let service = simple_service(move |request: String| {
    let query: ident::Query = request.parse().unwrap();
    let reply = query.process(&ip);

    // return future from reply
    futures::finished(reply.to_string())
});

let transport = new_ident_transport(stream);
pipeline::Server::new(service, transport)
```

Conclusion: Asynchronous I/O

- Leads to higher performance
- Rapidly evolving ecosystem
- Support in Rust at a turning point
- Next steps

Thank You!

Questions?

Code for the examples: https://github.com/matt2xu/rustfest2016

Bibliography

- About futures:
  - Introduction: https://aturon.github.io/blog/2016/08/11/futures/
  - Details about the design: https://aturon.github.io/blog/2016/09/07/futures-design/

- About Tokio:
  - Announcing Tokio: https://medium.com/@carllerche/announcing-tokio-df6bb4ddb34
  - Integration of Tokio with futures: http://aturon.github.io/blog/2016/08/26/tokio/

- Threads and processes on Linux:
  - http://stackoverflow.com/questions/807506/threads-vs-processes-in-linux

- Comparison of Web servers:
  - https://help.dreamhost.com/hc/en-us/articles/215945987-Web-server-performance-comparison