Next Gen Networking Infrastructure With Rust

BUOYANT

Hi, I’m @carllerche

You may remember me from...

Most newer databases are written in a language that includes a runtime

C / C++

Memory management

SEGV

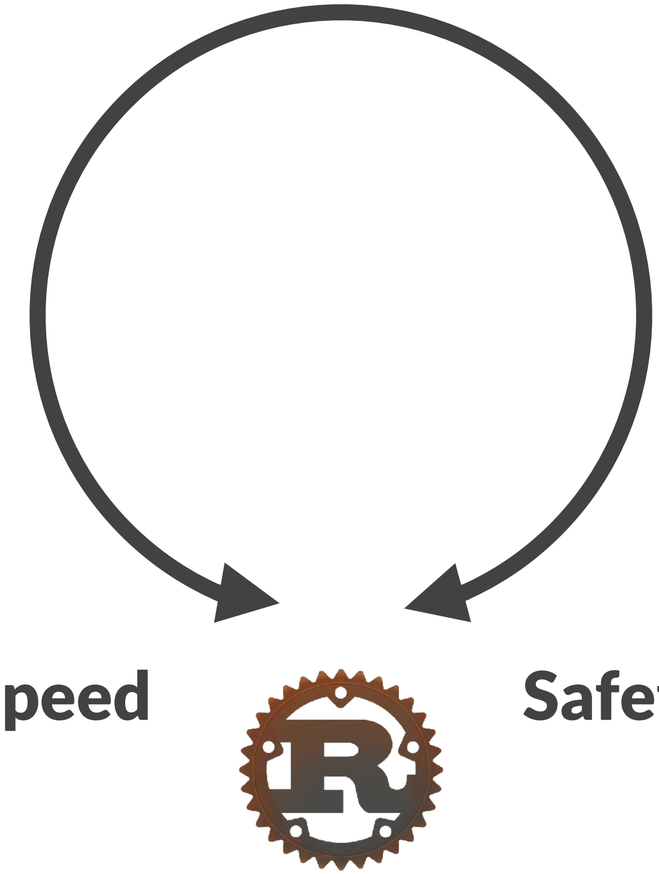
Heartbleed, cloudbleed, WannaCry

“Just use linting tools”

– Software developer who accidentally shipped remote code execution vulnerabilities to millions of computers


Speed
Safety


Speed
Safety


Speed
Safety

Checks at compile time

Ownership

Data has one owner

Compiles

```
fn print(s: String) {
    println!("{}", s);
}

let foo = String::new();
let bar = foo;

print(bar);
```

Compiles

```
fn print(s: String) {
    println!("{}", s);
}

let foo = String::new();
let bar = foo;

print(bar);
```

```
   |
8  |      let bar = foo;
   |          --- value moved here
9  |
10 |      print(foo);
   |            ^^^ value used here
after move
   |
   = note: move occurs because
`foo` has
     type `std::string::String`,
which does
     not implement the `Copy` trait
```

```
fn print(s: String) {
    println!("{}", s);
}

let foo = String::new();
let bar = foo;

print(foo);
```

Data is borrowed

Compiles

```
fn print(s: &String) {
    println!("{}", s);
}

let foo = String::new();
let bar = &foo;

print(&foo);
print(bar);
```

```
   |
10 | let bar = &foo;
   |           --- borrow of `foo`
occurs here
...
14 | drop(foo);
   |      ^^^ move out of `foo`
occurs here
```

```
fn print(s: &String) {
    println!("{}", s);
}

fn drop(s: String) {}

let foo = String::new();
let bar = &foo;

print(&foo);
drop(foo);
print(bar);
```

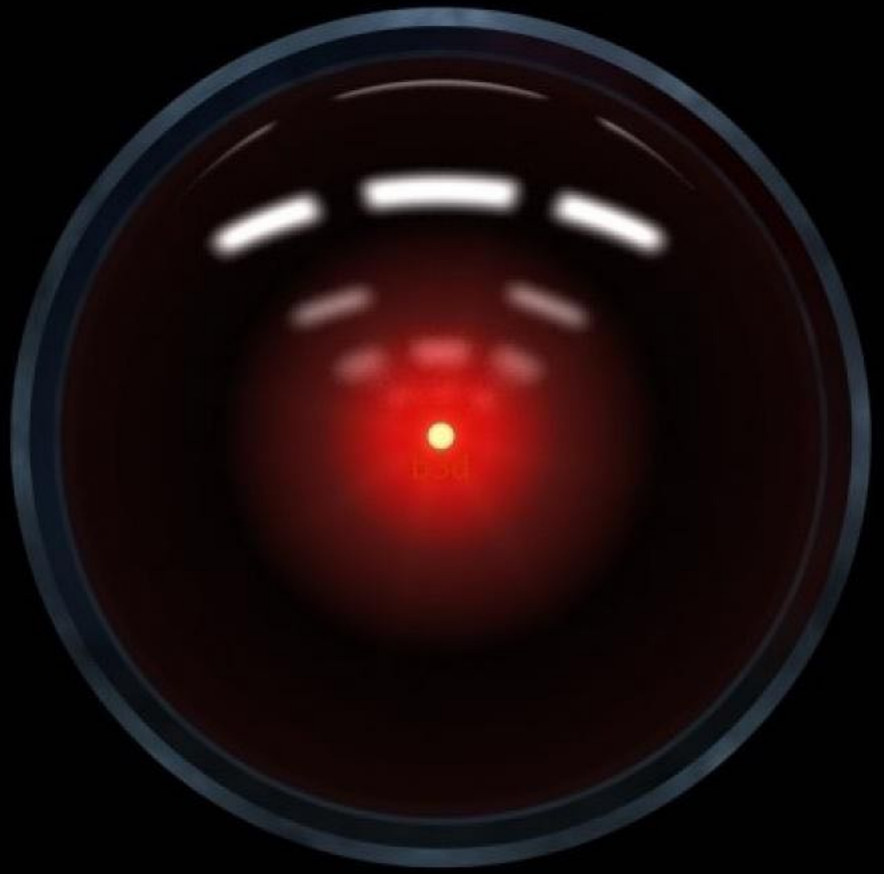
Mutable borrows

```
   |
8  | let bar = &mut foo;
   |                --- first
mutable borrow
   |                    occurs here
9  |
10 | print(&mut foo);
   |              ^^^ second mutable
borrow
   |                occurs here
11 | }
   | - first borrow ends here
```

```
fn print(s: &mut String) {
    println!("{}", s);
}

let mut foo = String::new();
let bar = &mut foo;

print(&mut foo);
```

```
   |
9  | let val = &vec[0];
   |               --- immutable borrow
occurs
   |                   here
...
13 | vec.push(String::new());
   | ^^^ mutable borrow occurs here
...
19 | }
   | - immutable borrow ends here
```

```
let mut vec = vec![];

vec.push(String::new());

// lots of code here

let val = &vec[0];

// lots of code here

vec.push(String::new());

// lots of code here

println!("{}", val);
```



I’m sorry, Dave. I’m afraid I can’t do that.

Modern Language Features

- Fearless concurrency
- Closures
- Type inference
- Traits
- Package manager
- Pattern matching
- Macros

Complexities of concurrent programming

- **What thread owns the data?**
- **What threads can read the data?**
- **What threads can mutate the data?**

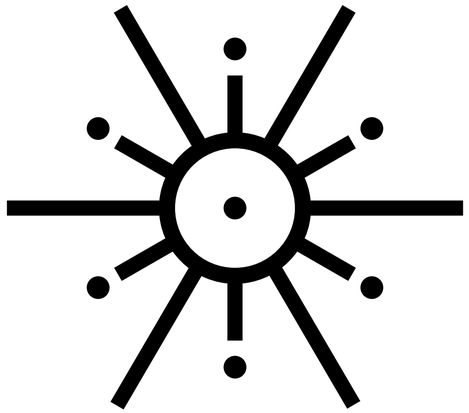
Message Passing

```
let (tx, rx) = mpsc::channel();

thread::spawn(move || {
    while let Ok(v) = rx.recv() {
        // Use var
    }
});

let val = "hello".to_string();
tx.send(val);

// Compile error
// println!("val={}", val);
```

Fast, Reliable, Productive

(pick three)

Can the principles of Rust be applied to a networking library?



Tokio

Fastest, Safest

Cancellation

Backpressure

@kanyewest
Kanye West

I hate when I'm on a flight and I wake up with a water bottle next to me like oh great now I gotta be responsible for this water bottle

16 Oct via web ☆ Favorite ↻ Retweet ↩ Reply

Secret: Do no work

```
let server = TcpListener::bind(&local_addr)?;

let server = sever.incoming().for_each(move |src| {
    let connection = TcpStream::connect(&remote_addr)
        .and_then(|move |dst| copy(src, dst));

    // Run asynchronously in the background
    task::spawn(connection);
    Ok(())
});

task::spawn(server);
```

Futures

```
TcpStream::connect(&remote_addr, &handle)
    .and_then(move |dst| { … })
    .and_then(|foo| { … })
```

```
let fut_1 = copy(src_rd, dst_wr);

let fut_2 = copy(dst_rd, src_wr);

let fut_3 = fut_1.join(fut_2);
```

Zero cost

Epoll

- **Non-blocking sockets**
- **Event queue**

```
let socket = bind(remote_addr);
epoll.register(socket, token: 0);
let clients = State::new();

for event in epoll.poll():
  match event.token:
    0 =>
      while socket.is_ready():
        let client = socket.accept();
        let token = clients.store(client);
        epoll.register(client, token: token);

    token =>
      let client = clients[token];
      process(client);
```

1. Connect a socket
2. Send handshake message
3. Receive handshake message
4. Send request
5. Receive response

```
enum SocketState {
    Connecting,
    SendingHandshake,
    ReceivingHandshake,
    SendingRequest,
    ReceivingResponse,
}
```

Fast

- No runtime allocations
- No dynamic dispatch
- No copying / growing the stack
- No garbage collection

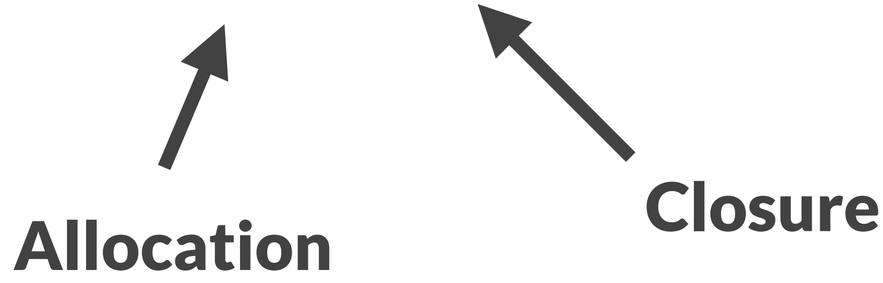
After compilation, Tokio is equivalent.

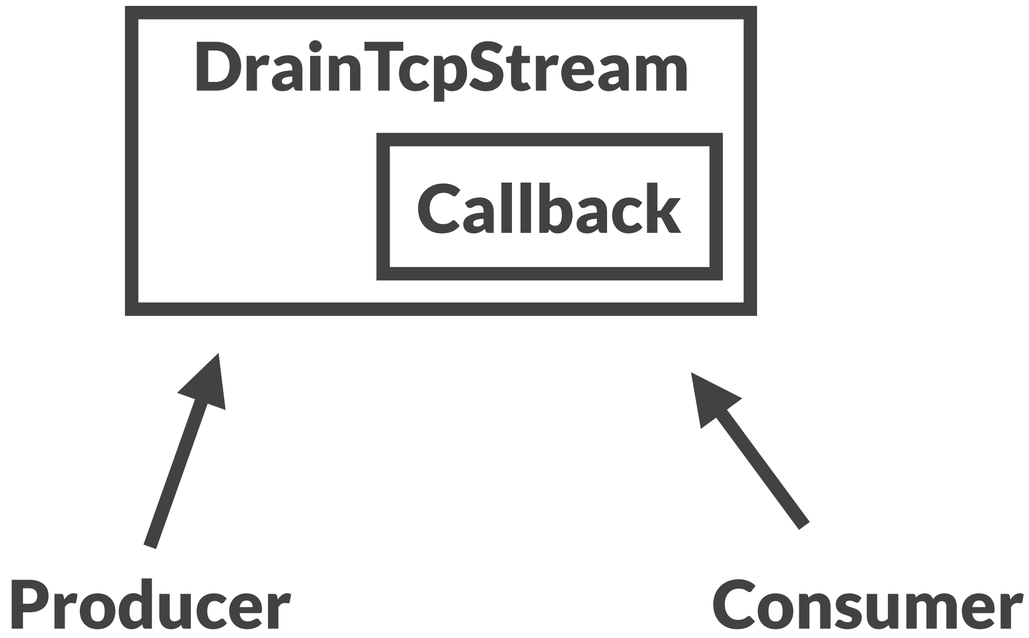
Fast

- No runtime allocations
- No dynamic dispatch
- No copying / growing the stack
- No garbage collection

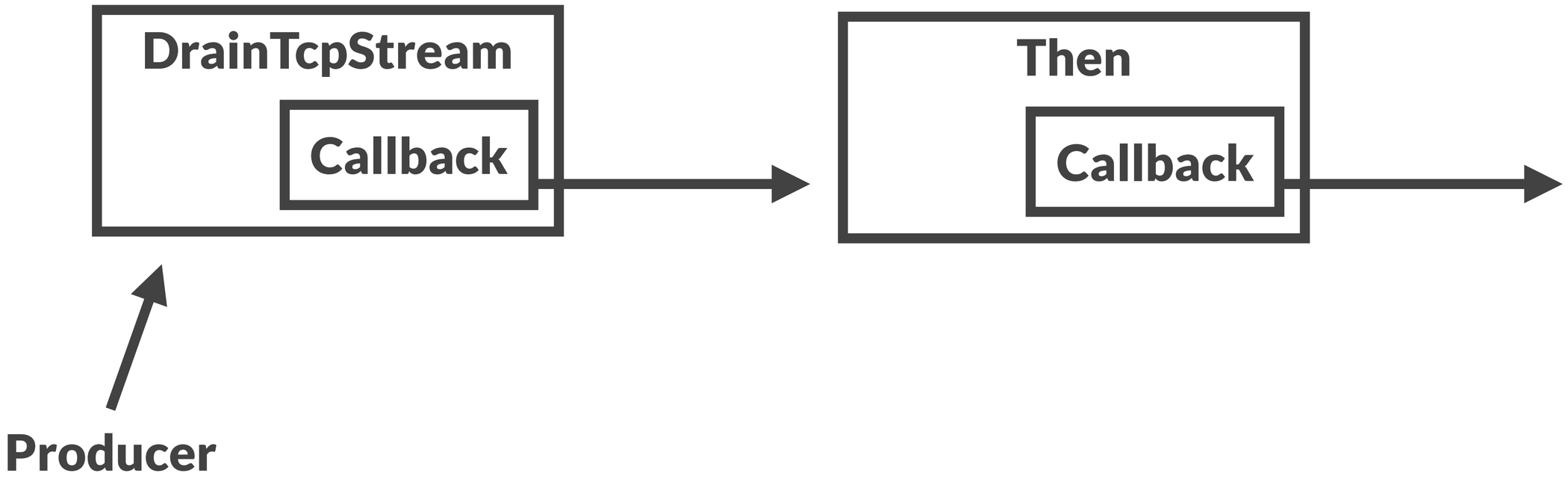
Pull, not push

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

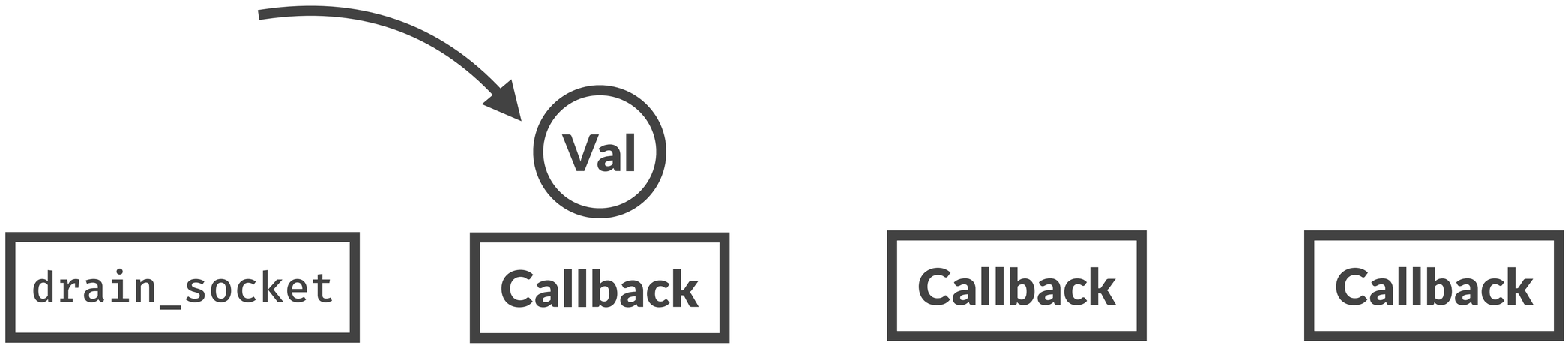


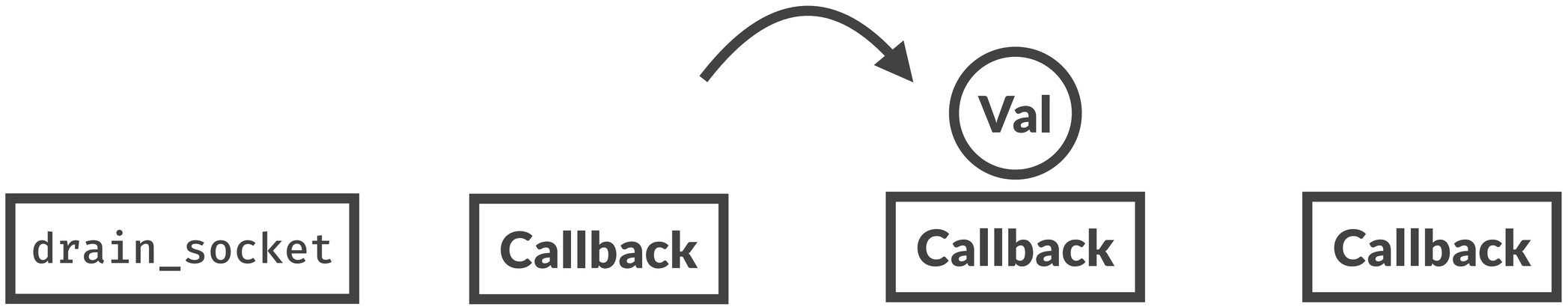

DrainTcpStream
Callback
Producer
Consumer

```
drain_socket
  .then(...)
  .then(...)
  .then(...)
```


DrainTcpStream
Callback
Then
Callback
Producer

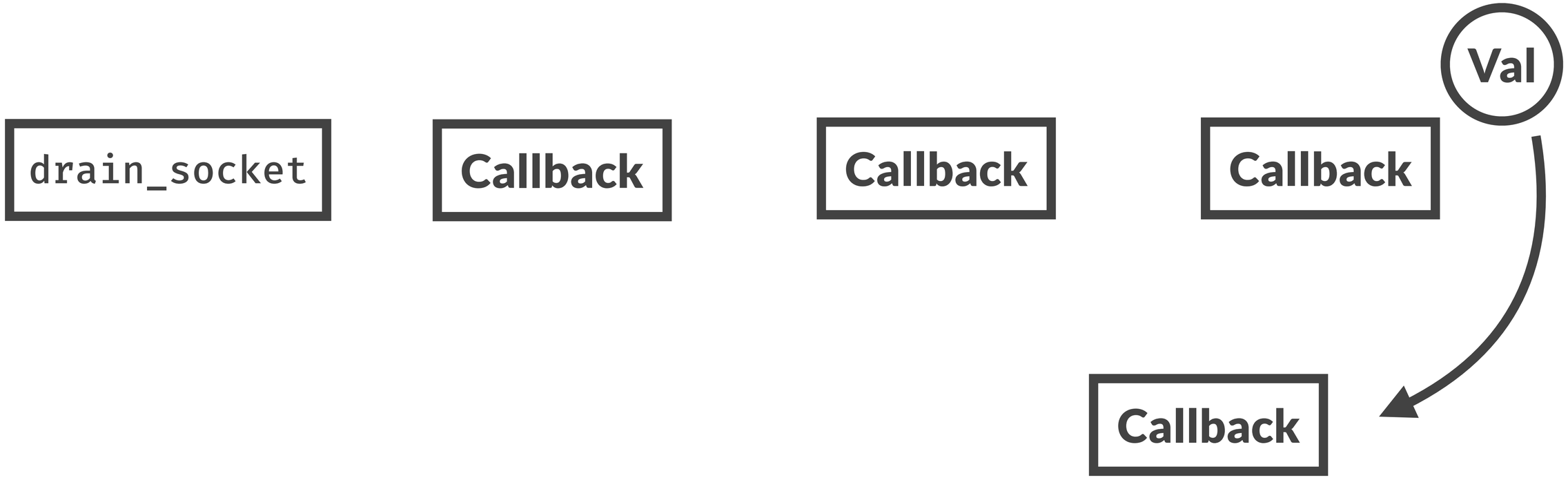

Val
drain_socket
Callback
Callback
Callback

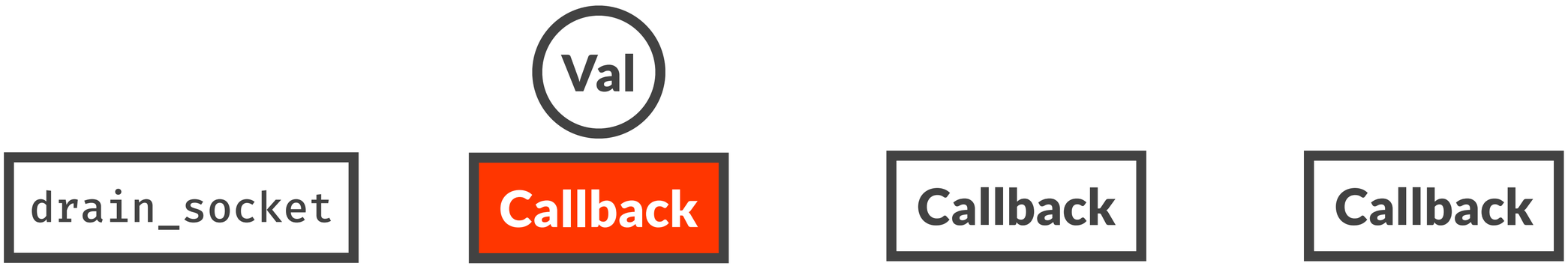

Val
drain_socket
Callback
Callback
Callback

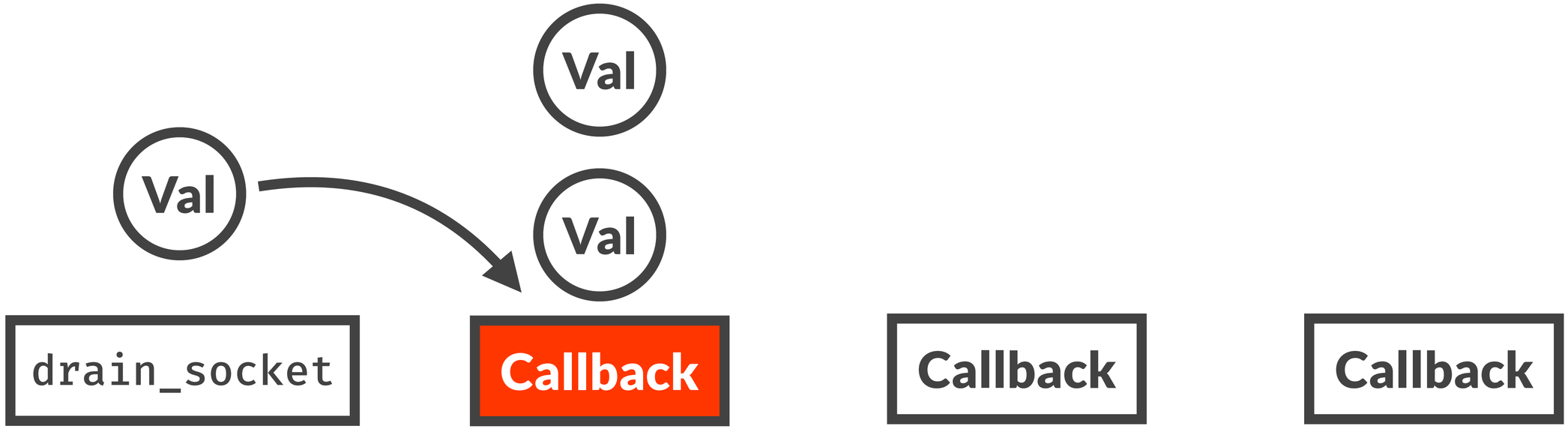

Val
drain_socket
Callback
Callback
Callback


drain_socket
Callback
Callback
Val
Callback


drain_socket
Callback
Callback
Callback
Val

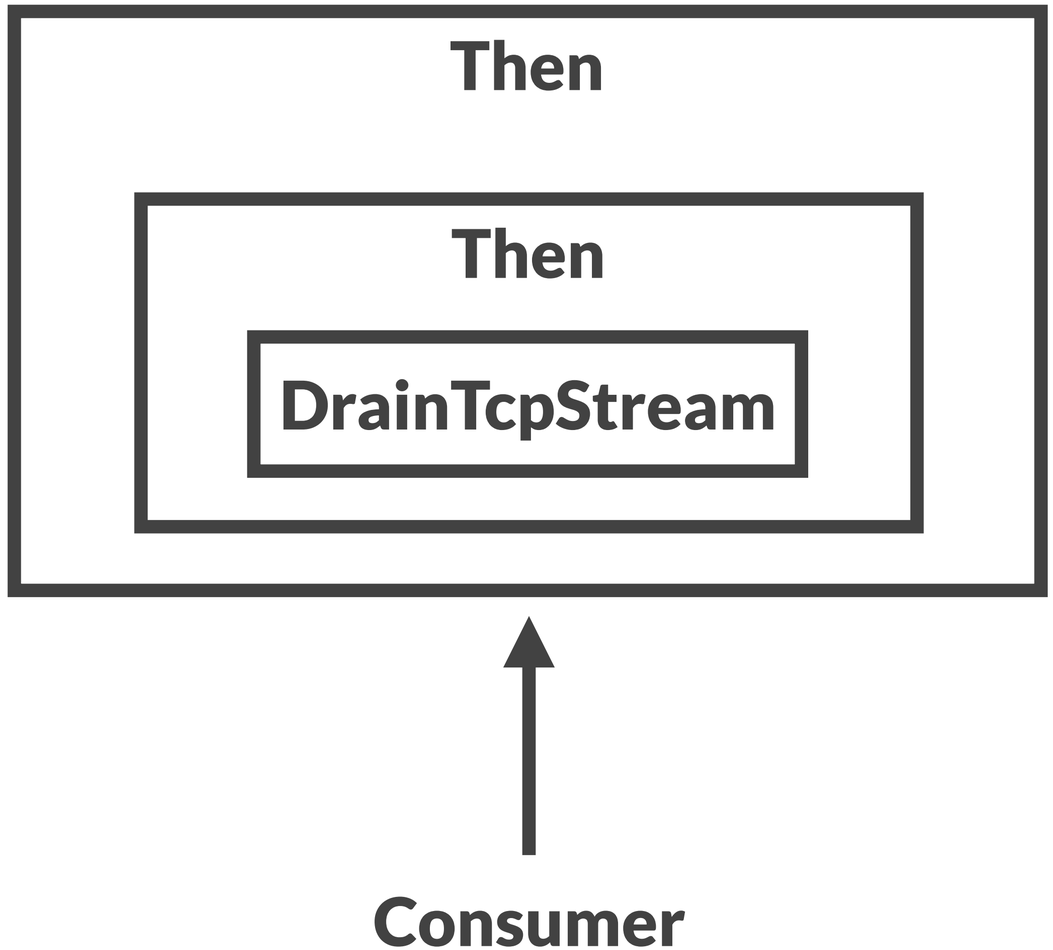

drain_socket
Callback
Callback
Callback
Val
Callback


Val
drain_socket
Callback
Callback
Callback

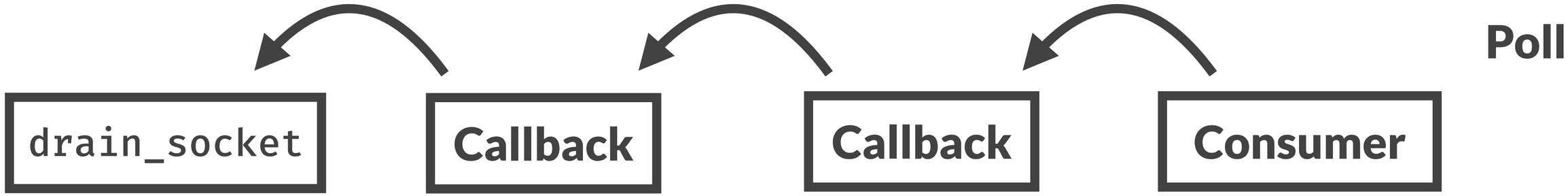

Val
Val
Val
drain_socket
Callback
Callback
Callback

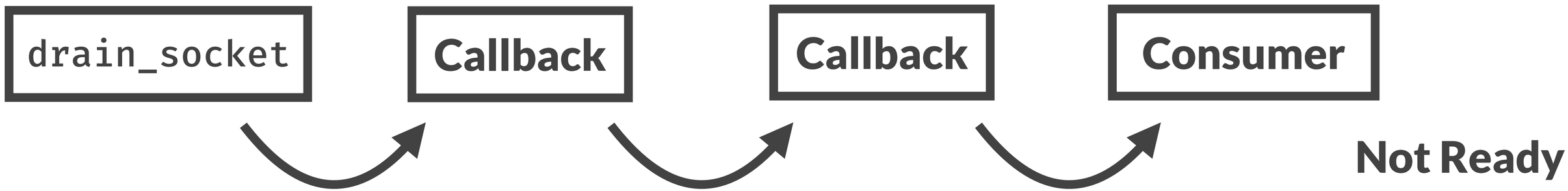
```
struct DrainTcpStream {
  socket: TcpStream,
  nread: u64,
}
```

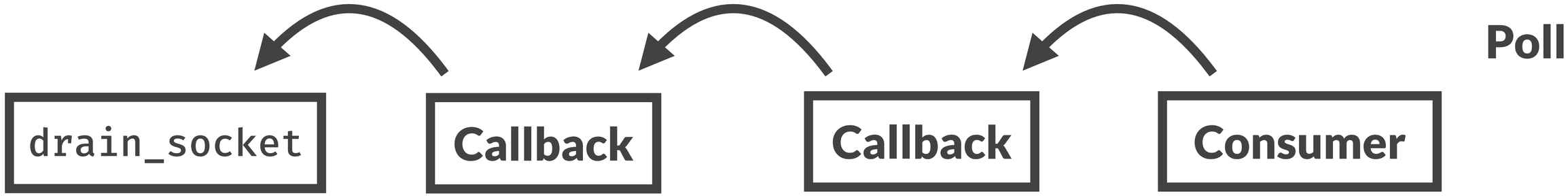
```
drain_socket
  .then(...)
  .then(...)
  .then(...)
```

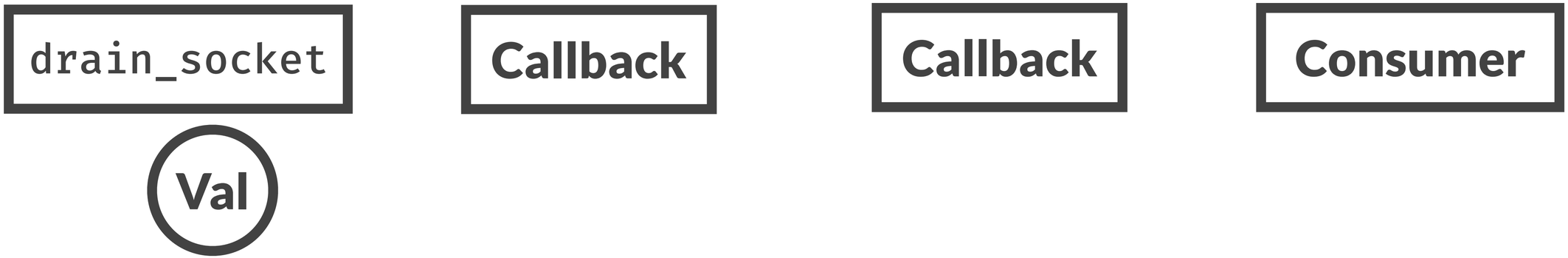

Then
Then
DrainTcpStream
Consumer

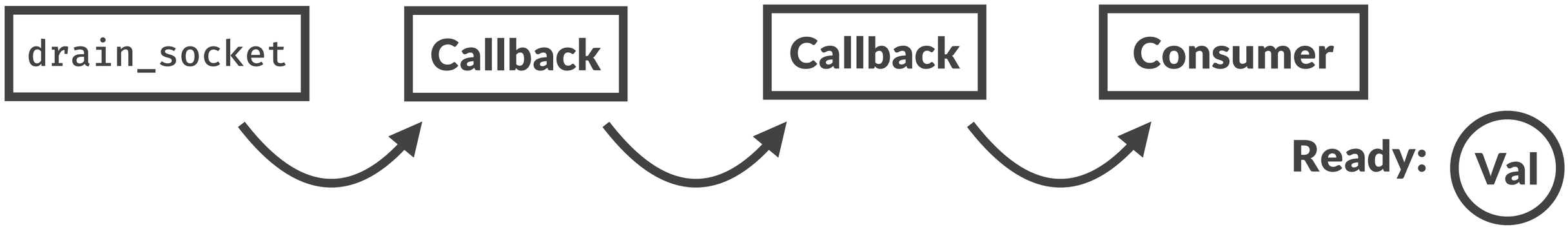
```
fn poll(&mut self) -> Async<Bytes> {
  let mut buf = [0; 1024];
  loop {
    match self.socket.read(&mut buf) {
      Ok(0) => return Async::Ready(self.nread),
      Ok(n) => self.nread += n,
      Err(e) => return Async::NotReady,
    }
  }
}
```


drain_socket
Callback
Callback
Consumer
Poll

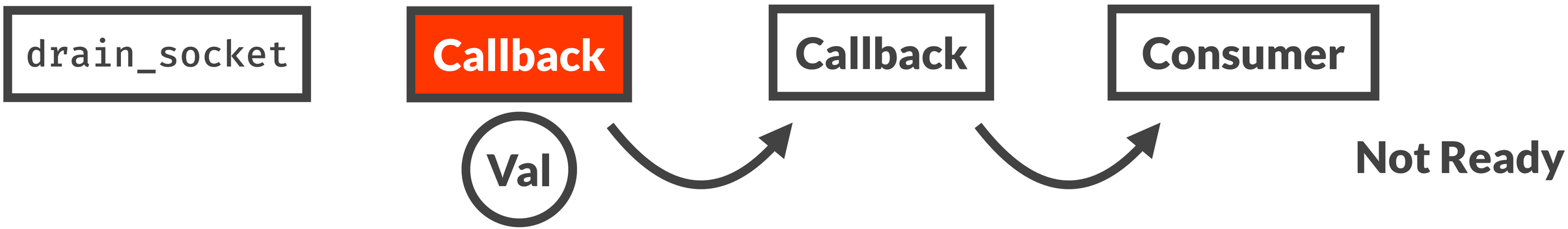

drain_socket
Callback
Callback
Consumer
Not Ready

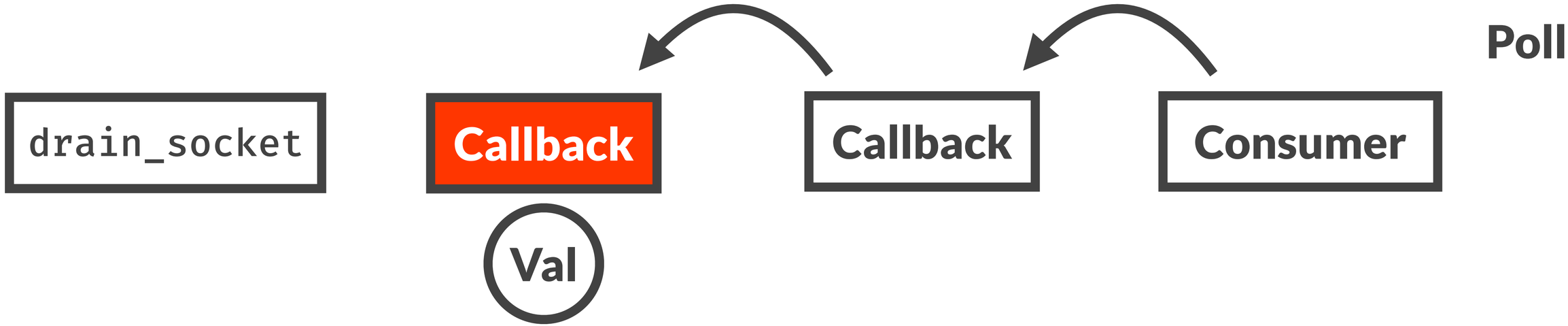

drain_socket
Callback
Callback
Consumer
Poll

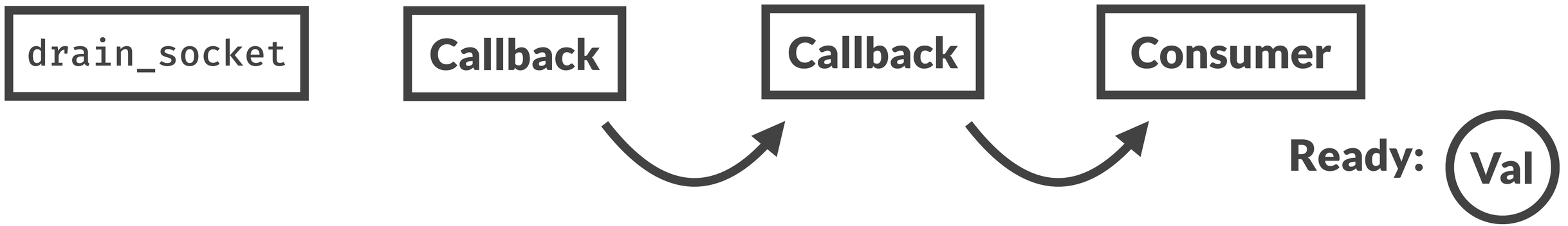

drain_socket
Callback
Callback
Consumer
Val


drain_socket
Callback
Callback
Consumer
Ready:
Val


drain_socket
Callback
Val
Callback
Consumer


drain_socket
Callback
Callback
Consumer
Val
Not Ready


drain_socket
Callback
Val
Callback
Consumer
Poll


drain_socket
Callback
Callback
Consumer
Ready:
Val

Cancellation is just drop

```
enum Then<A, B, F> {
    First(A, F),
    Second(B),
}

fn poll(&mut self) -> Async<B::Item> {
    loop {
        let fut_b = match *self {
            Then::First(ref mut fut_a, ref f) => {
                match fut_a.poll() {
                    Async::Ready(v) => f(v),
                    Async::NotReady => Async::NotReady,
                }
            }
            Then::Second(ref mut fut_b) => return fut_b.poll(),
        }

        *self = Then::Second(fut_b);
    }
}
```

1. Connect a socket

2. Send handshake message

3. Receive handshake message

4. Send request

5. Receive response

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

1. Connect a socket
2. Send handshake message
3. Receive handshake message
4. Send request
5. Receive response

```
enum SocketState {
    Connecting,
    SendingHandshake,
    ReceivingHandshake,
    SendingRequest,
    ReceivingResponse,
}
```

Have your 🍰 and 🍽️ it too.

Tokio + Rust gets you speed and safety

Thanks!

https://tokio.rs

https://linkerd.io