Thinking Asynchronously: Designing Applications with Boost.Asio

Chris Kohlhoff

The Basics

The Basics

```
asio::io_service io_service;

// ...

tcp::socket socket(io_service);

// ...

socket.async_connect(
        server_endpoint,
        your_completion_handler);

// ...

io_service.run();
```

```
asio::io_service io_service;

// ...

tcp::socket socket(io_service);

// ...

socket.async_connect(
        server_endpoint,
        your_completion_handler);

// ...

io_service.run();
```

```
asio::io_service io_service;

// ...

tcp::socket socket(io_service);

// ...

socket.async_connect(
        server_endpoint,
        your_completion_handler);

// ...

io_service.run();
```

I/O Object

```
asio::io_service io_service;

// ...

tcp::socket socket(io_service);

// ...

socket.async_connect(
        server_endpoint,
        your_completion_handler);

// ...

io_service.run();
```

Asynchronous operation

```
asio::io_service io_service;

// ...

tcp::socket socket(io_service);

// ...

socket.async_connect(
        server_endpoint,
        your_completion_handler);

// ...

io_service.run();
```

The Basics

```
socket.async_connect(
        server_endpoint,
        your_completion_handler);
```

```
socket.async_connect(
        server_endpoint,
        your_completion_handler);
```

I/O Object

```
socket.async_connect(
        server_endpoint,
        your_completion_handler);
```

Initiating function

```
socket.async_connect(
        server_endpoint,
        your_completion_handler);
```

Arguments

```
socket.async_connect(
        server_endpoint,
        your_completion_handler);
```

Completion handler

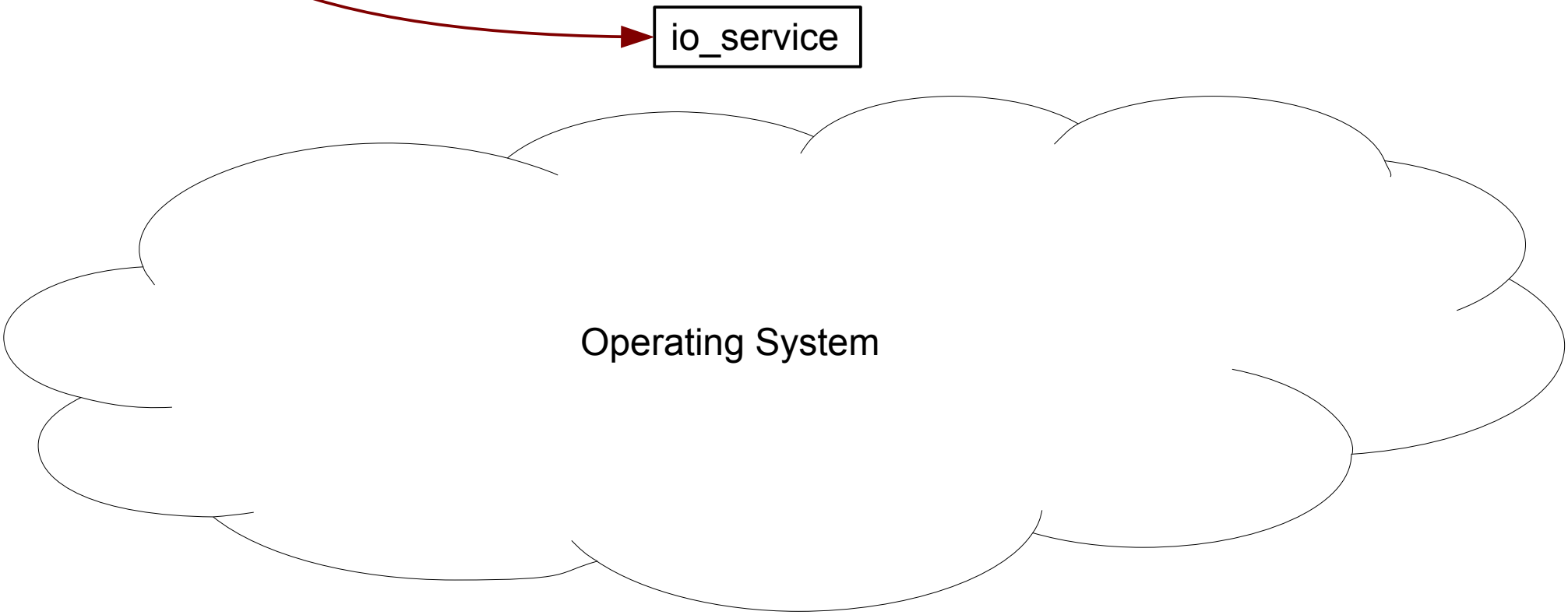
```
socket.async_connect(
        server_endpoint,
        your_completion_handler);
```

Completion handler

```
void your_completion_handler(
        const boost::system::error_code& ec);
```

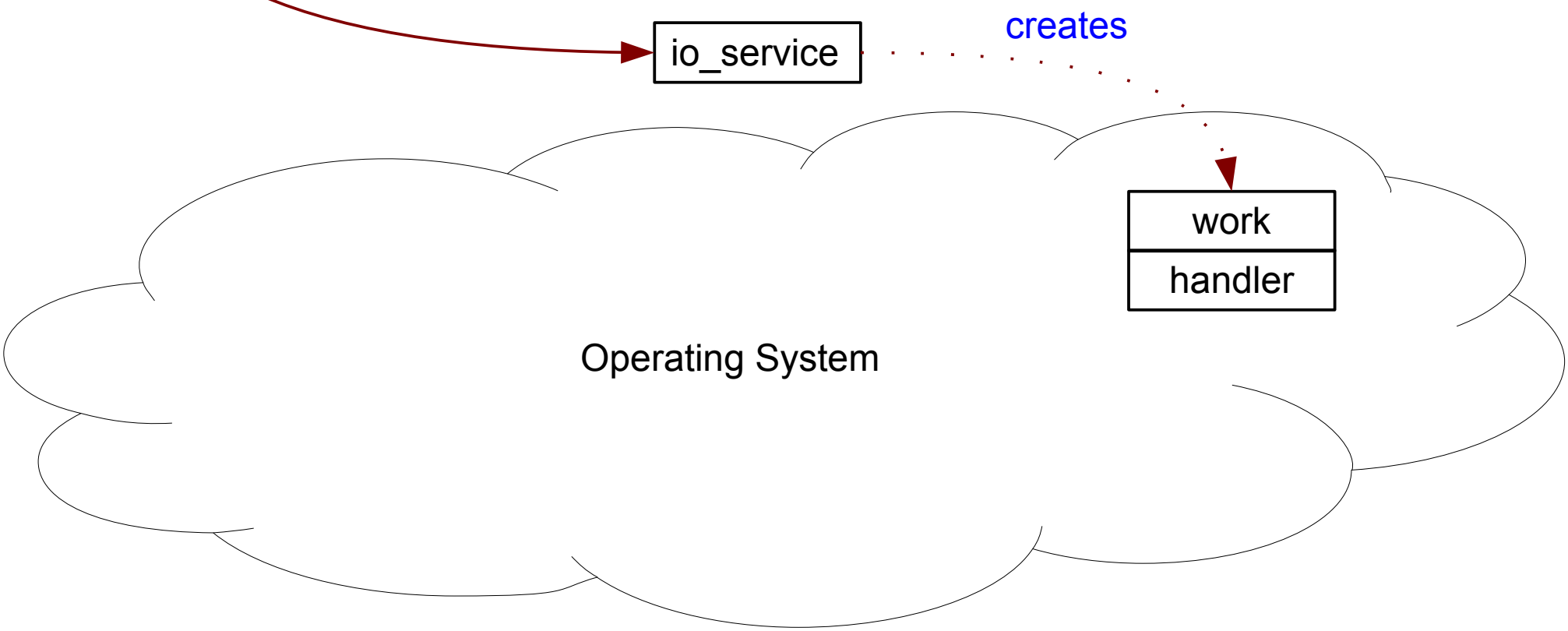
The Basics

```
socket.async_connect(
    server_endpoint,
    your_completion_handler);
```

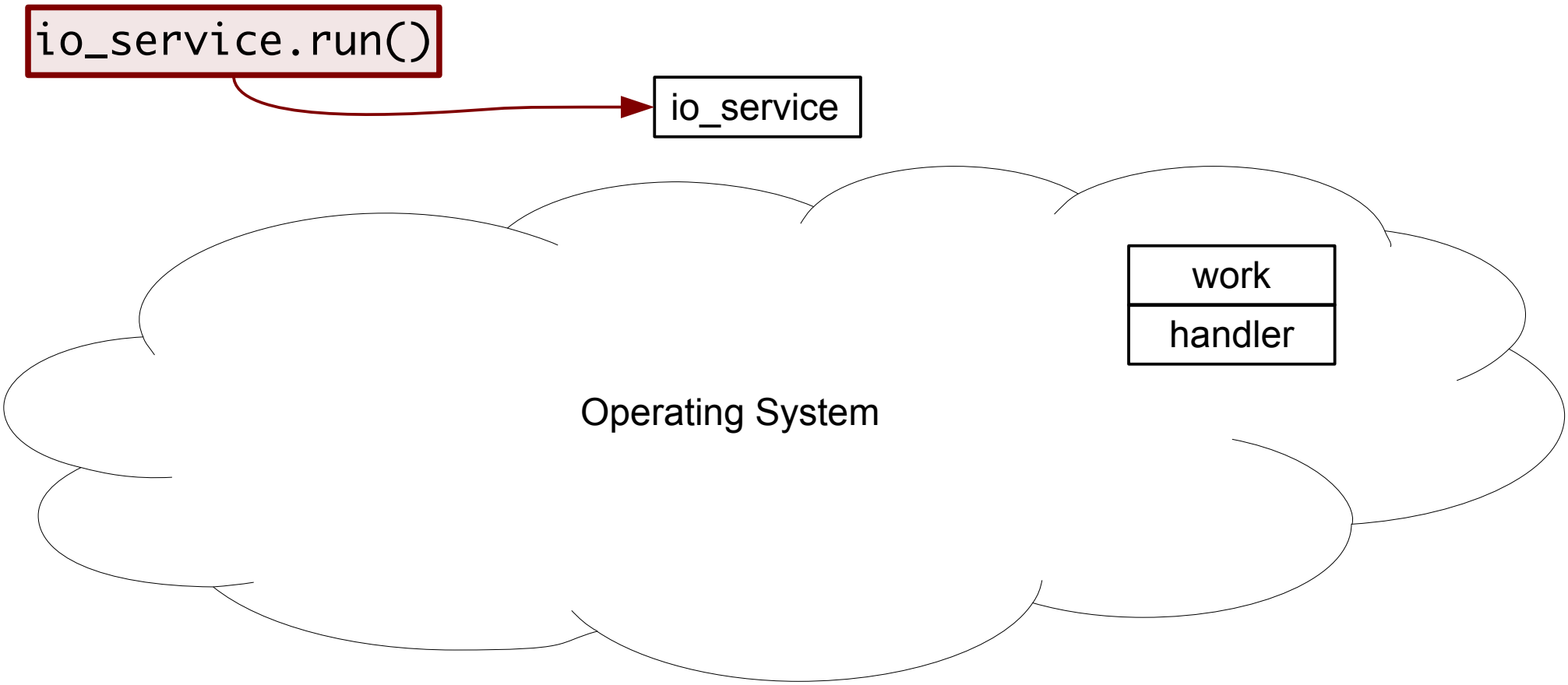


The Basics

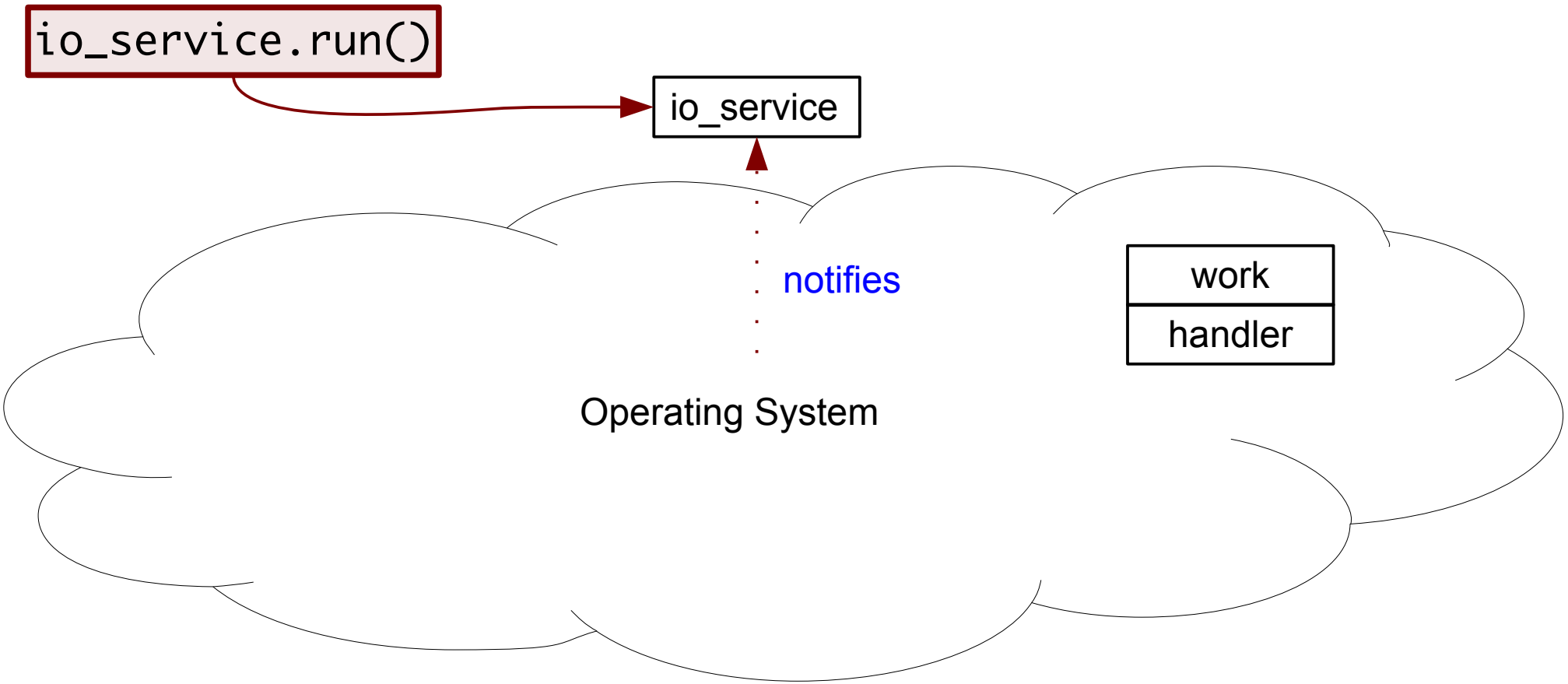
```
socket.async_connect(
    server_endpoint,
    your_completion_handler);
```



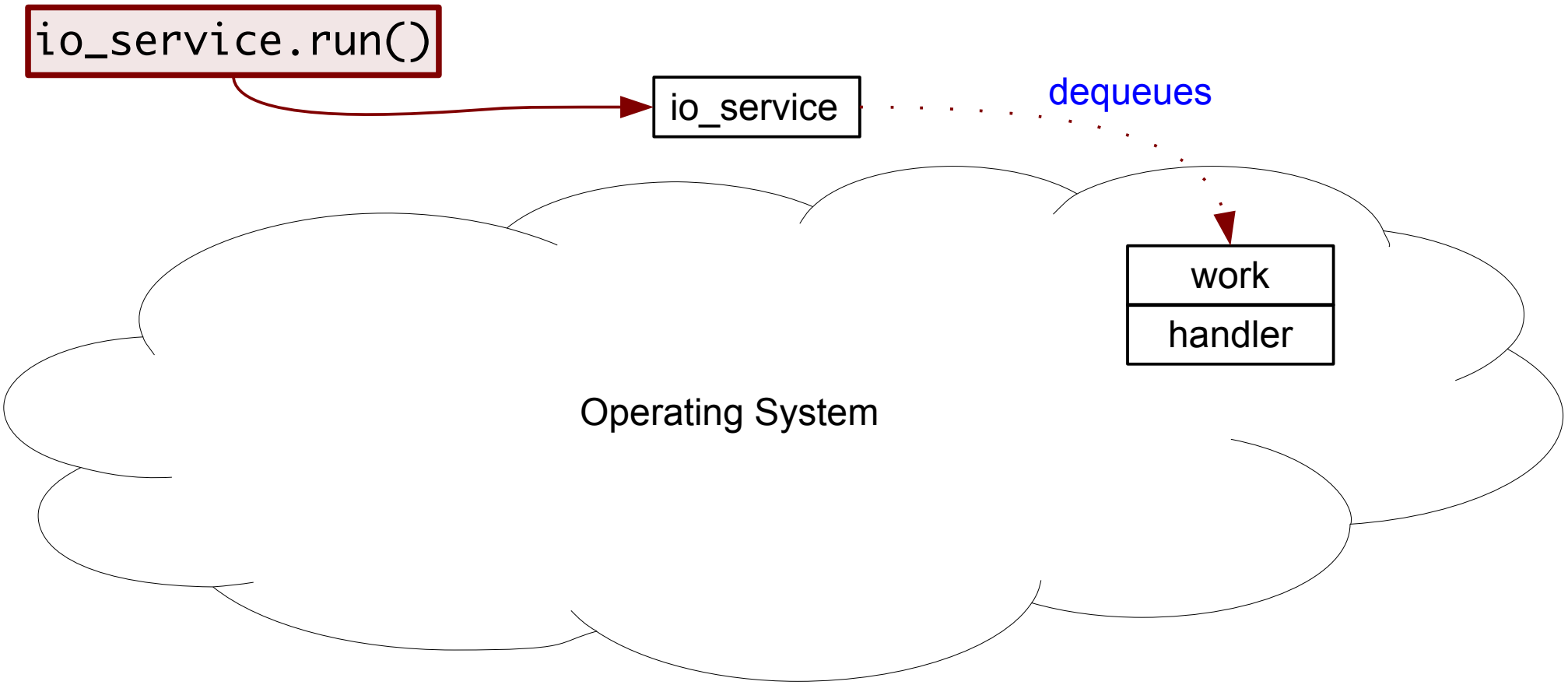
The Basics



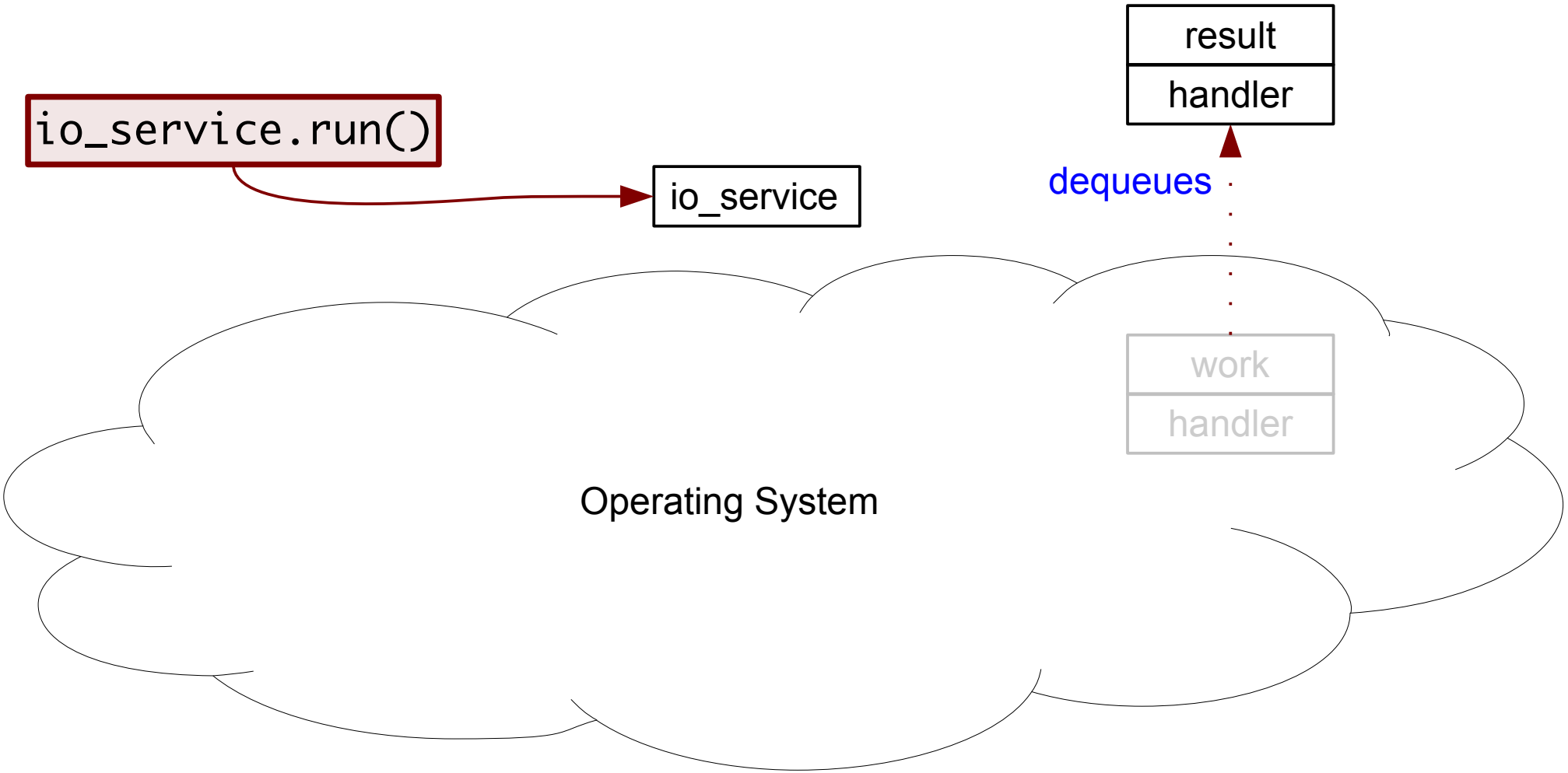
The Basics


io_service.run()
io_service
notifies
work
handler
Operating System

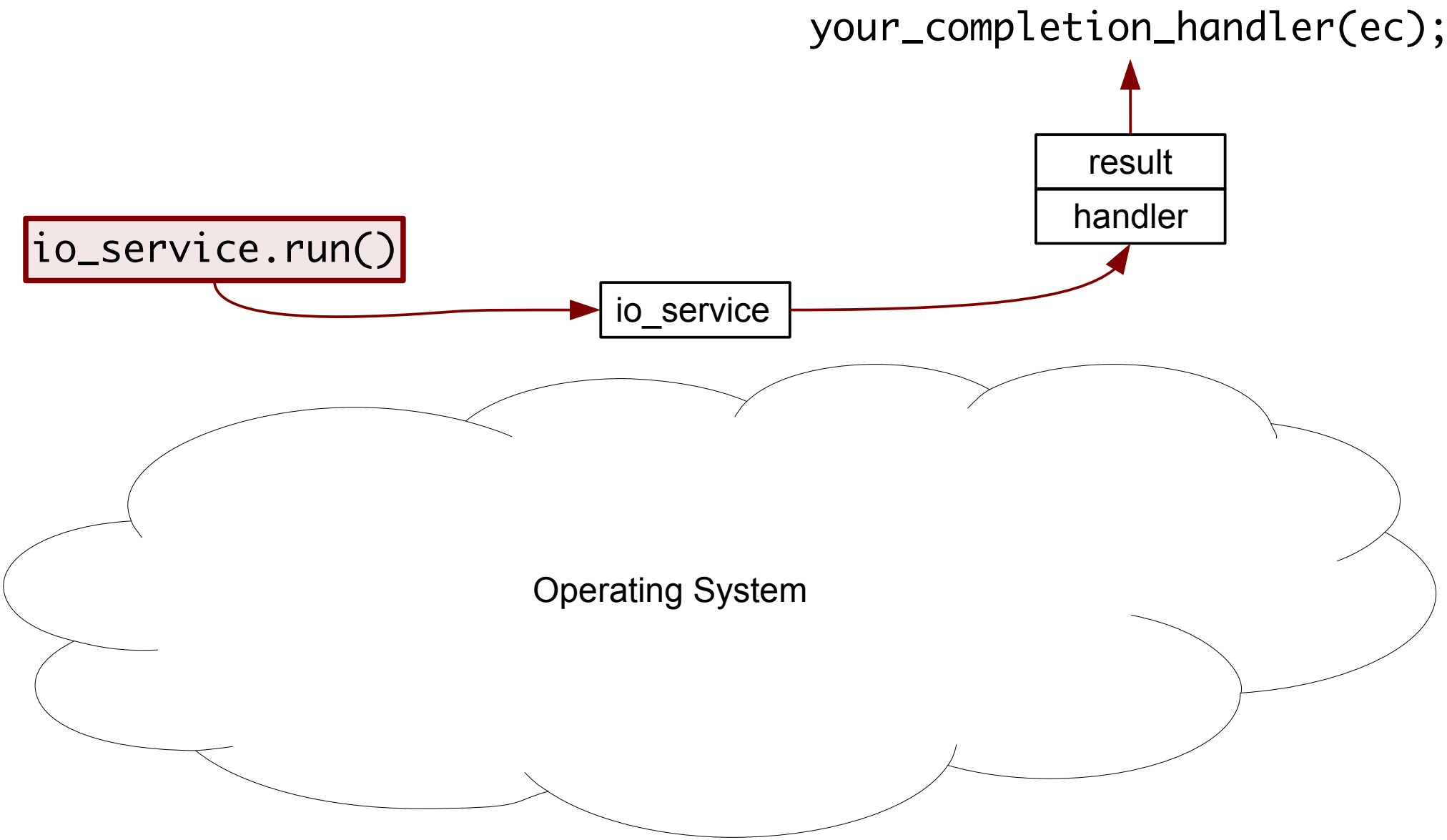
The Basics


io_service.run()
io_service
dequeues
work
handler
Operating System

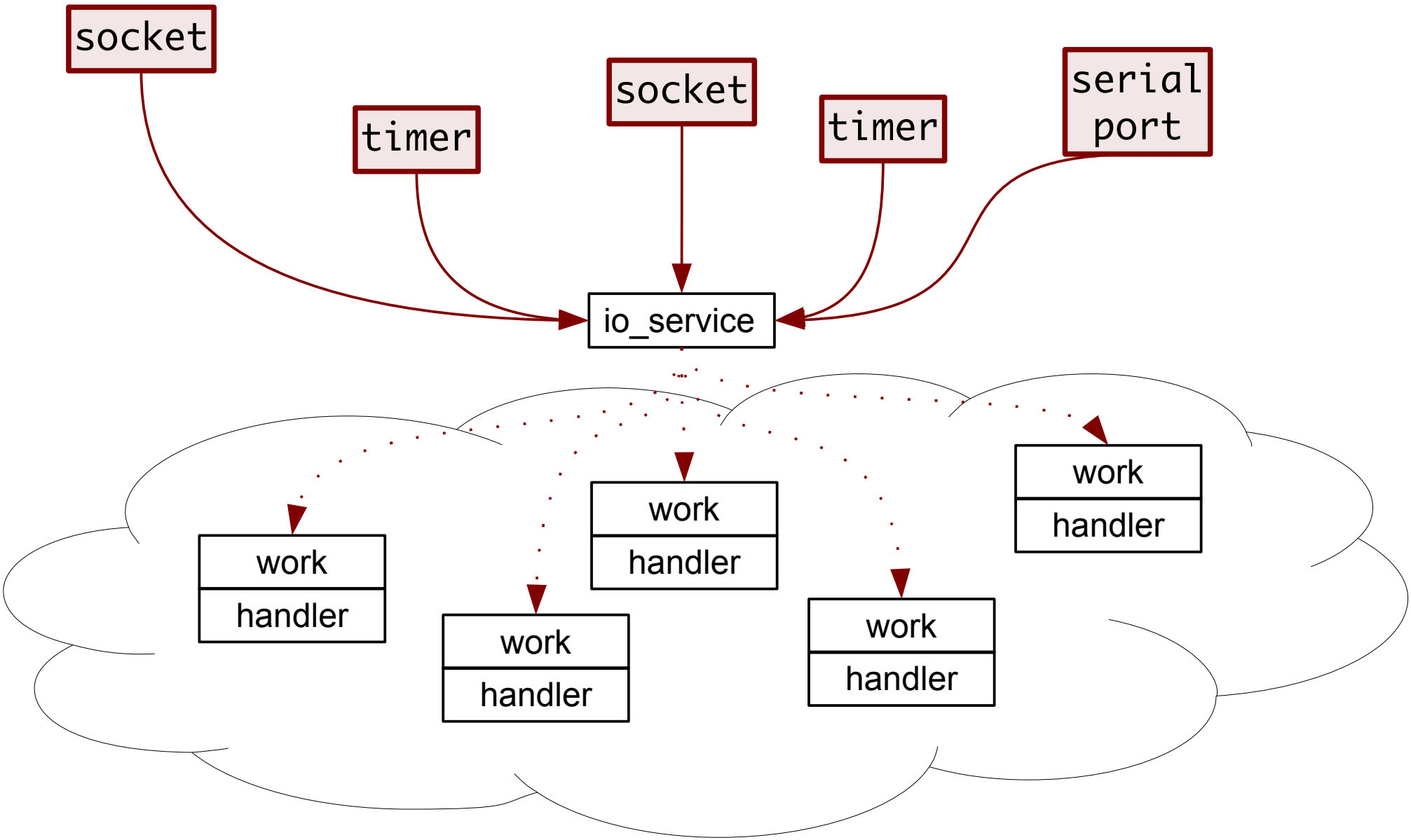
The Basics


io_service.run()
io_service
result
handler
dequeues
work
handler
Operating System

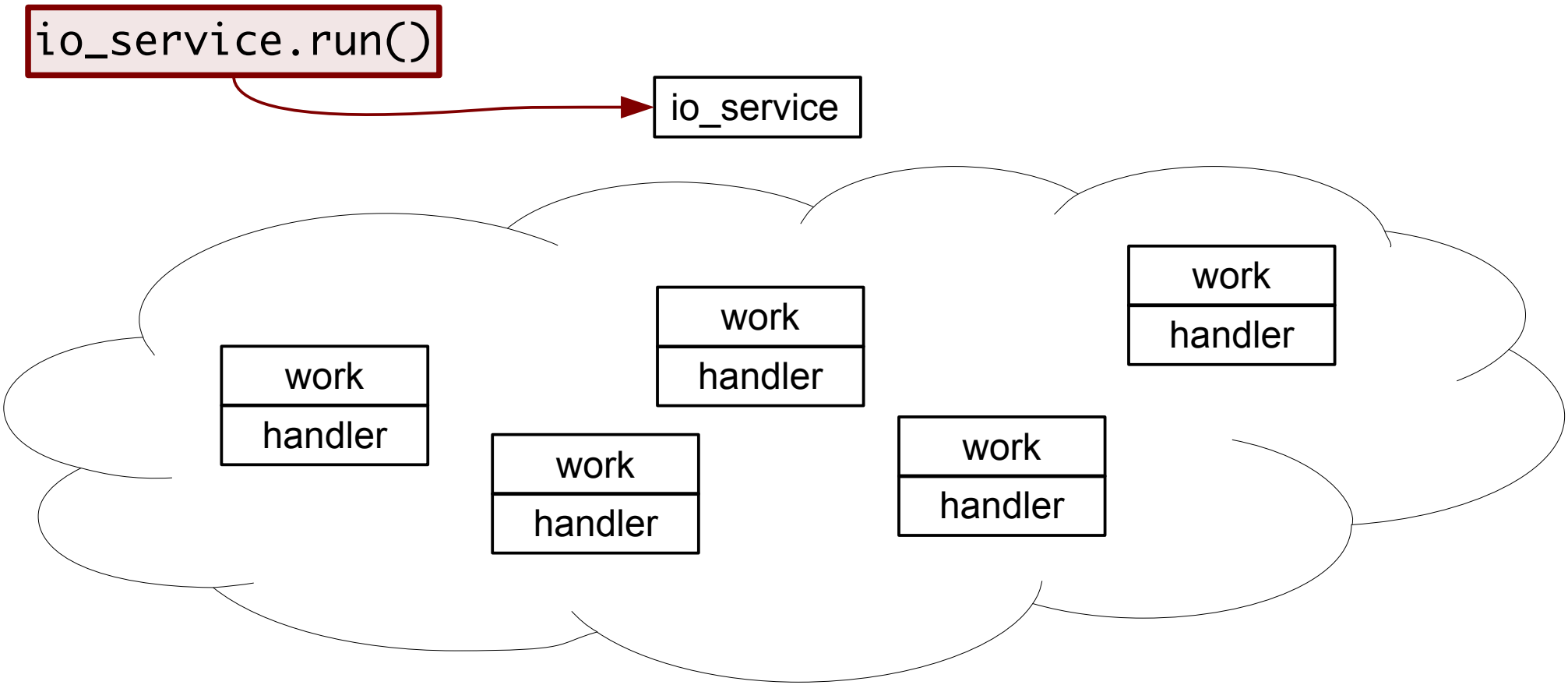
The Basics


your_completion_handler(ec);
result
handler
io_service.run()
io_service
Operating System

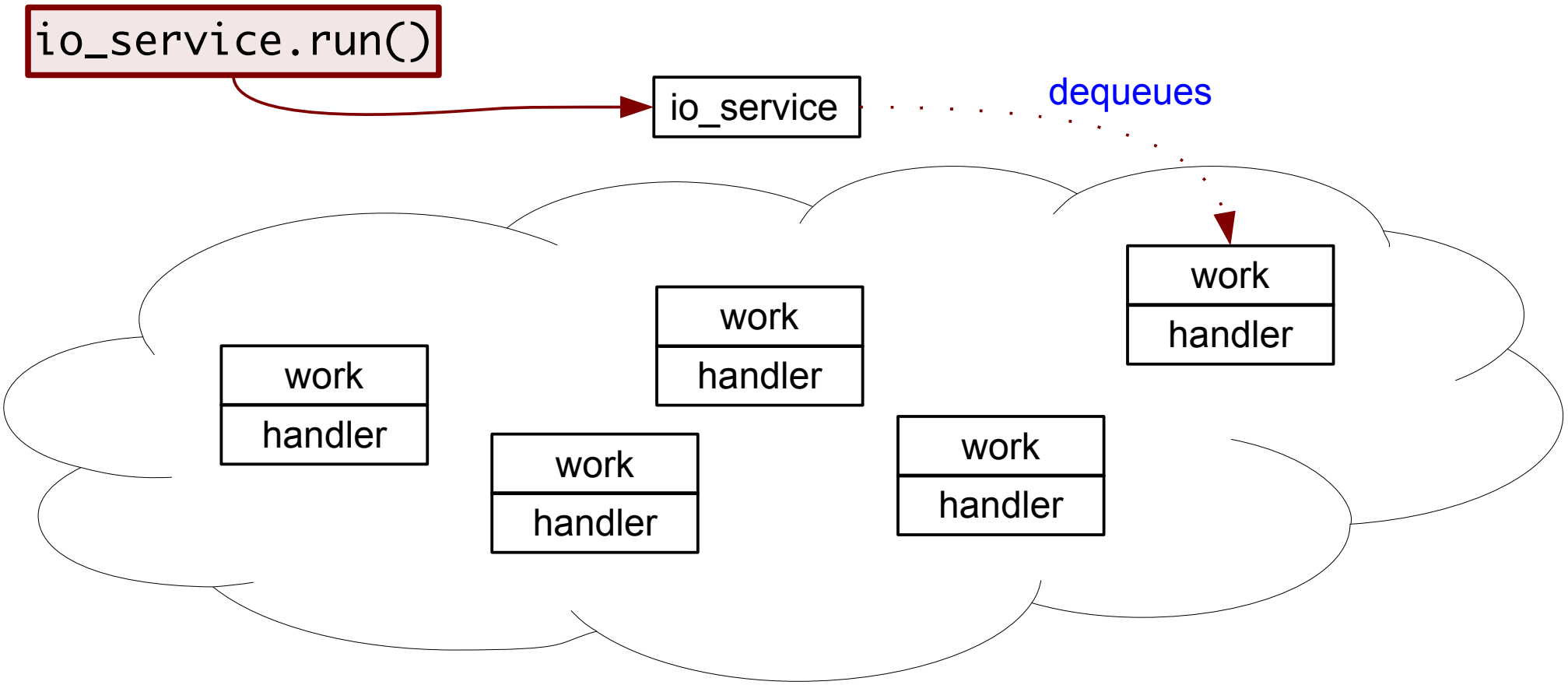
The Basics


socket
timer
socket
timer
serial port
io_service
work
handler
work
handler
work
handler
work
handler
work
handler

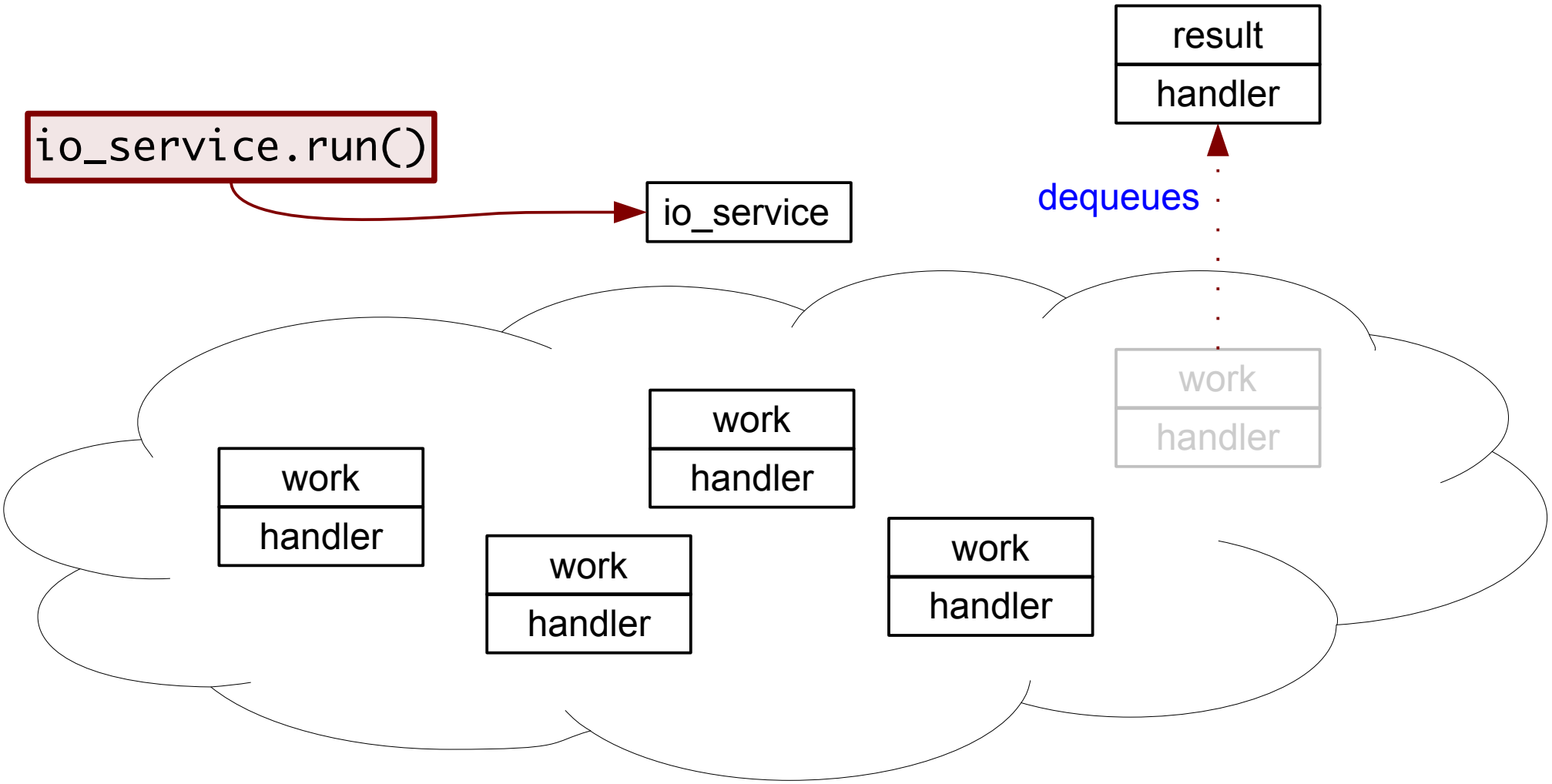
The Basics


io_service.run()
io_service
work
handler
work
handler
work
handler
work
handler
work
handler

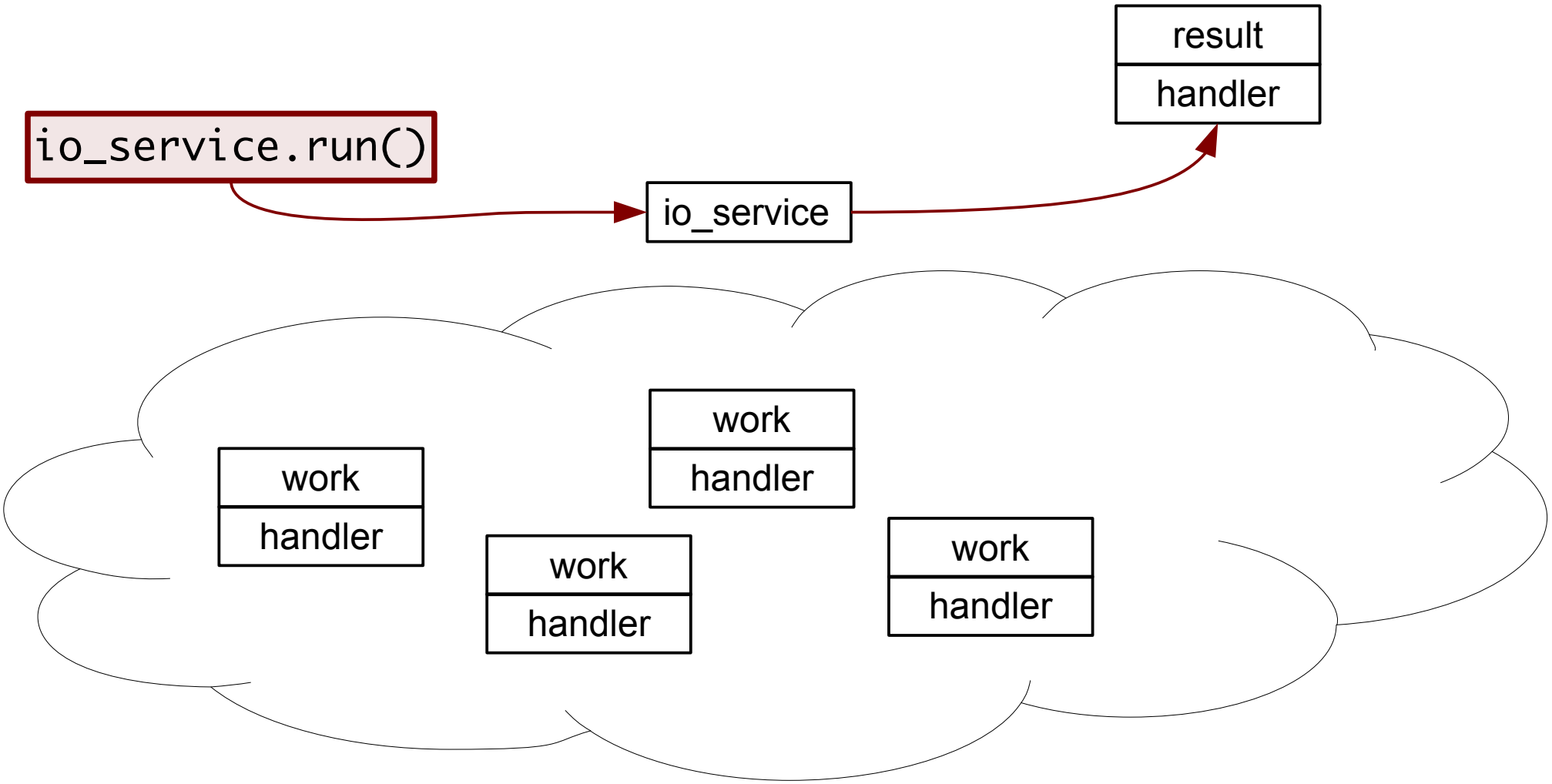
The Basics


io_service.run()
io_service
dequeues
work
handler
work
handler
work
handler
work
handler
work
handler

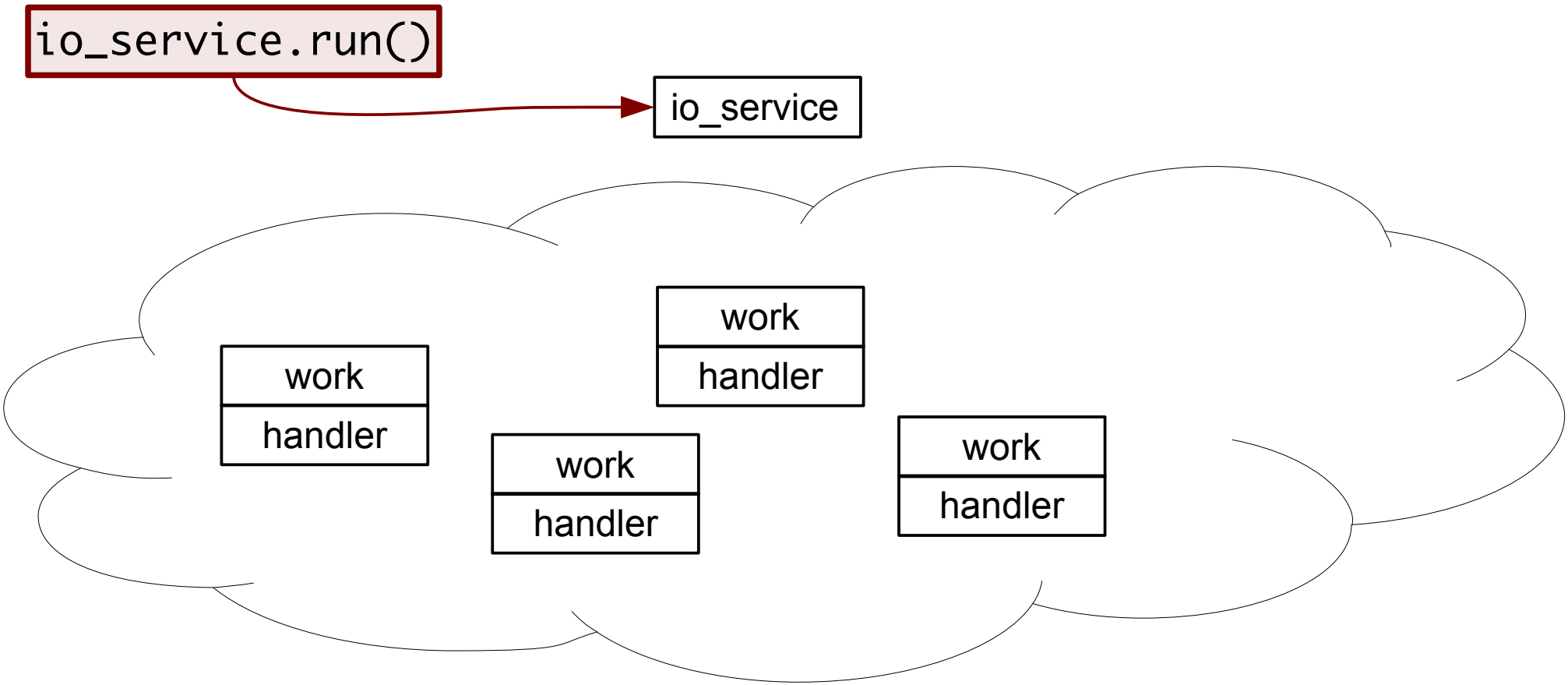
The Basics


io_service.run()
io_service
result
handler
dequeues
work
handler
work
handler
work
handler
work
handler
work
handler

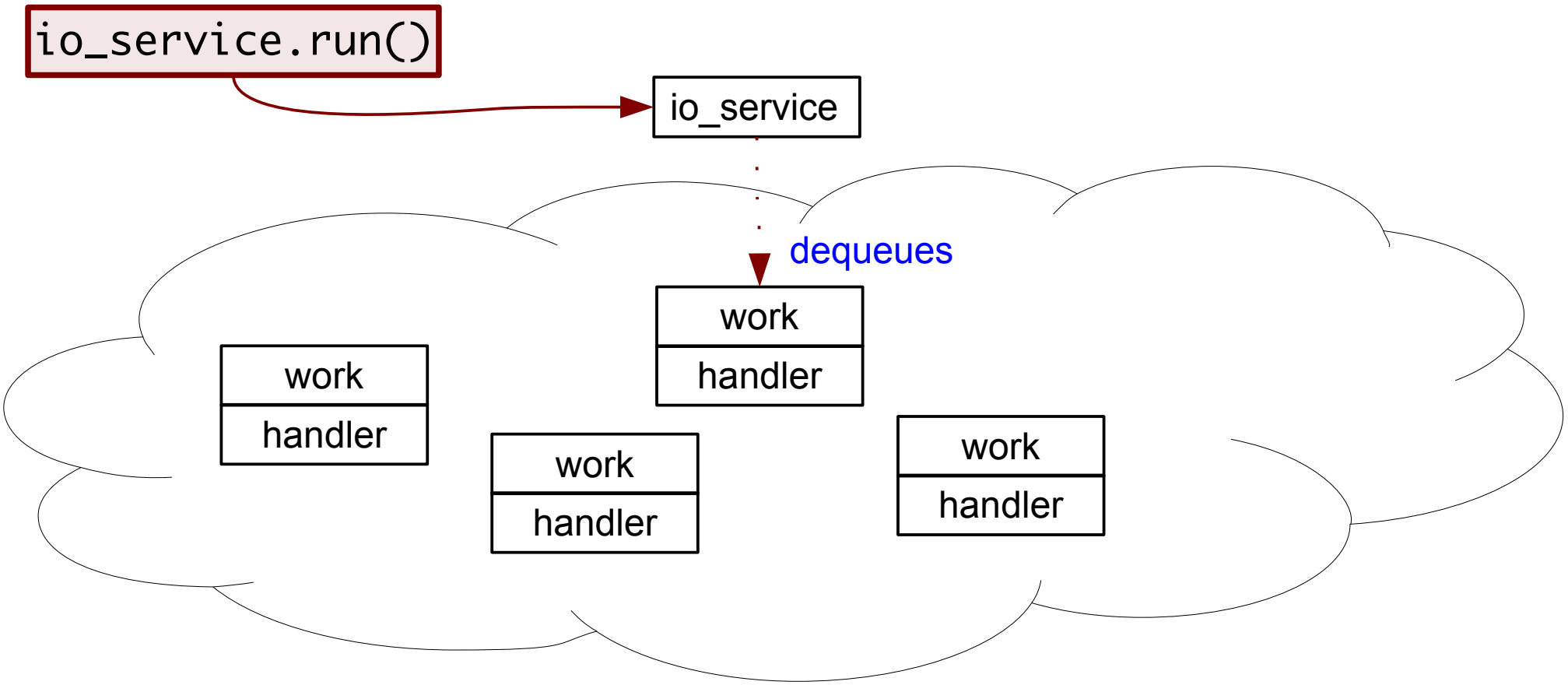
The Basics


io_service.run()
io_service
result
handler
work
handler
work
handler
work
handler
work
handler

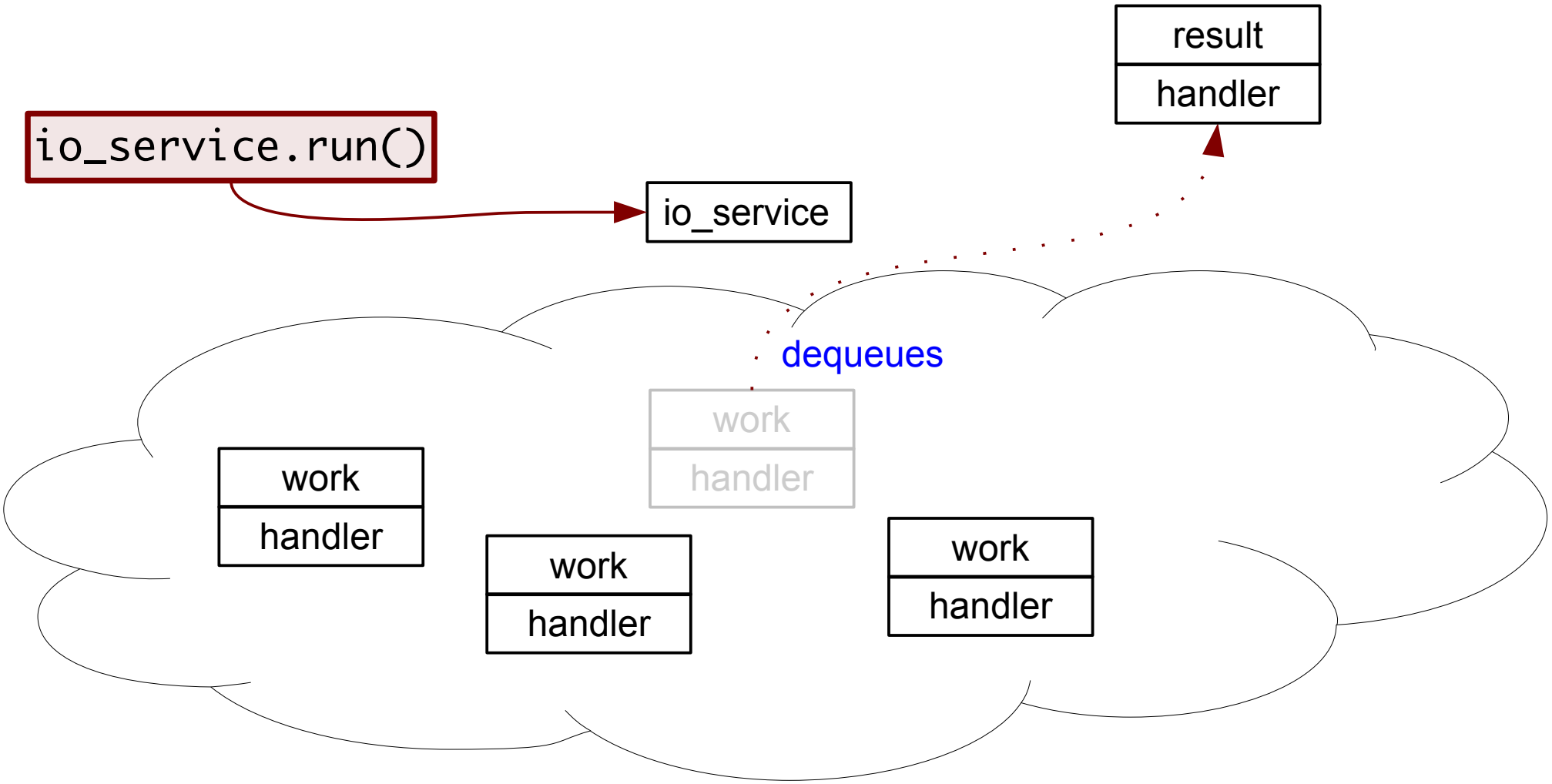
The Basics


io_service.run()
io_service
work
handler
work
handler
work
handler
work
handler

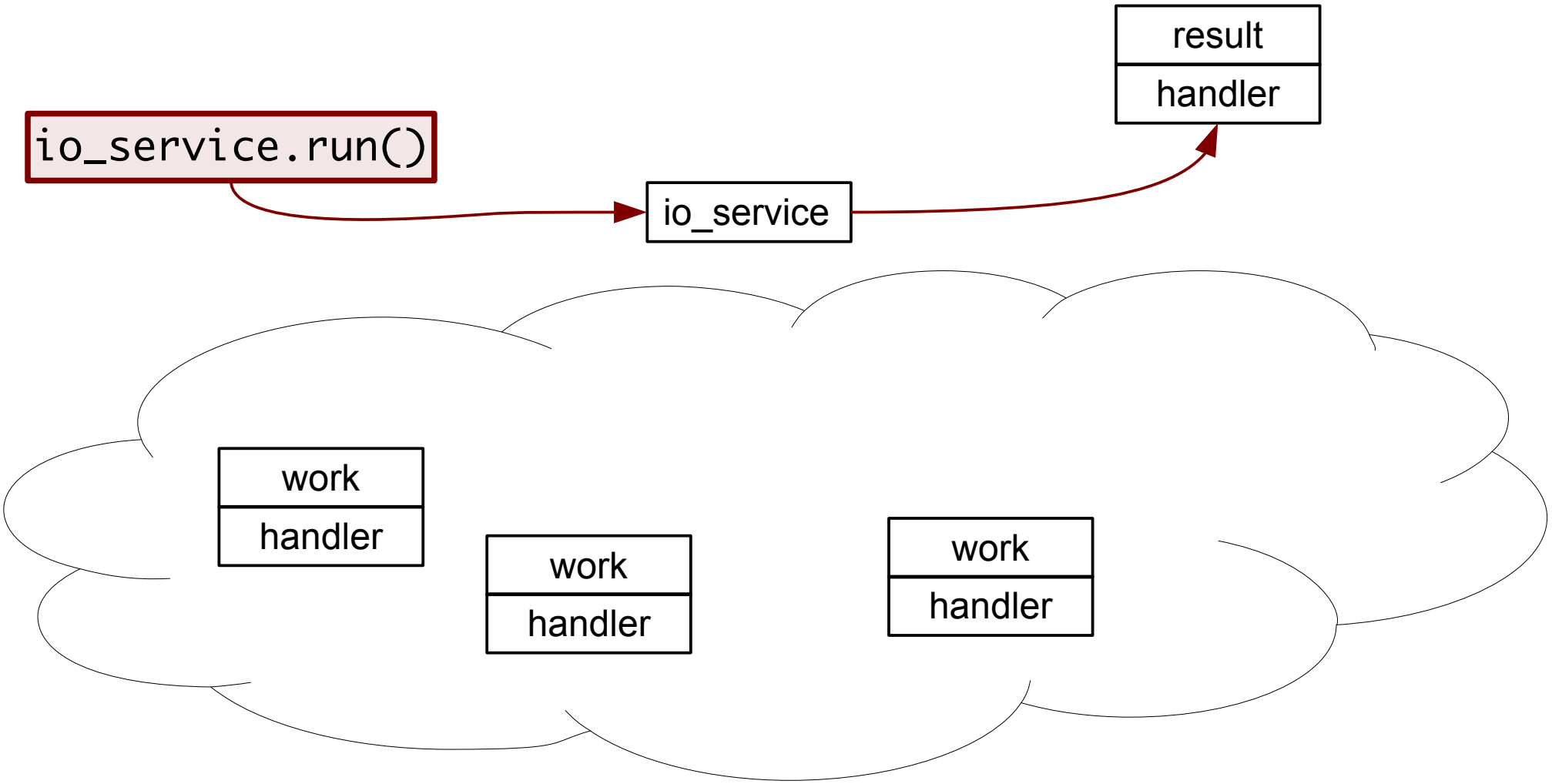
The Basics


io_service.run()
io_service
dequeues
work
handler
work
handler
work
handler
work
handler

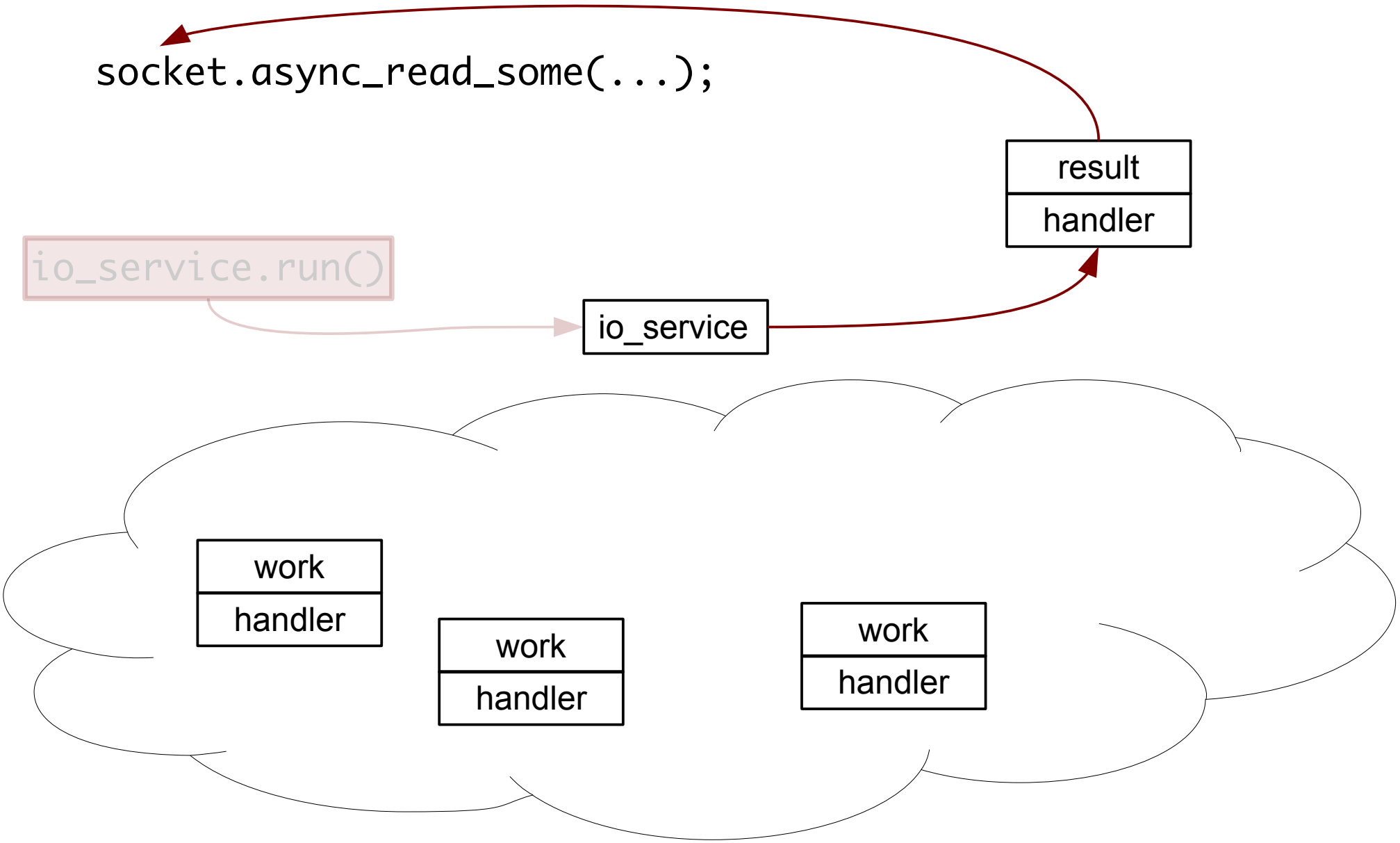
The Basics


io_service.run()
io_service
result
handler
dequeues
work
handler
work
handler
work
handler
work
handler

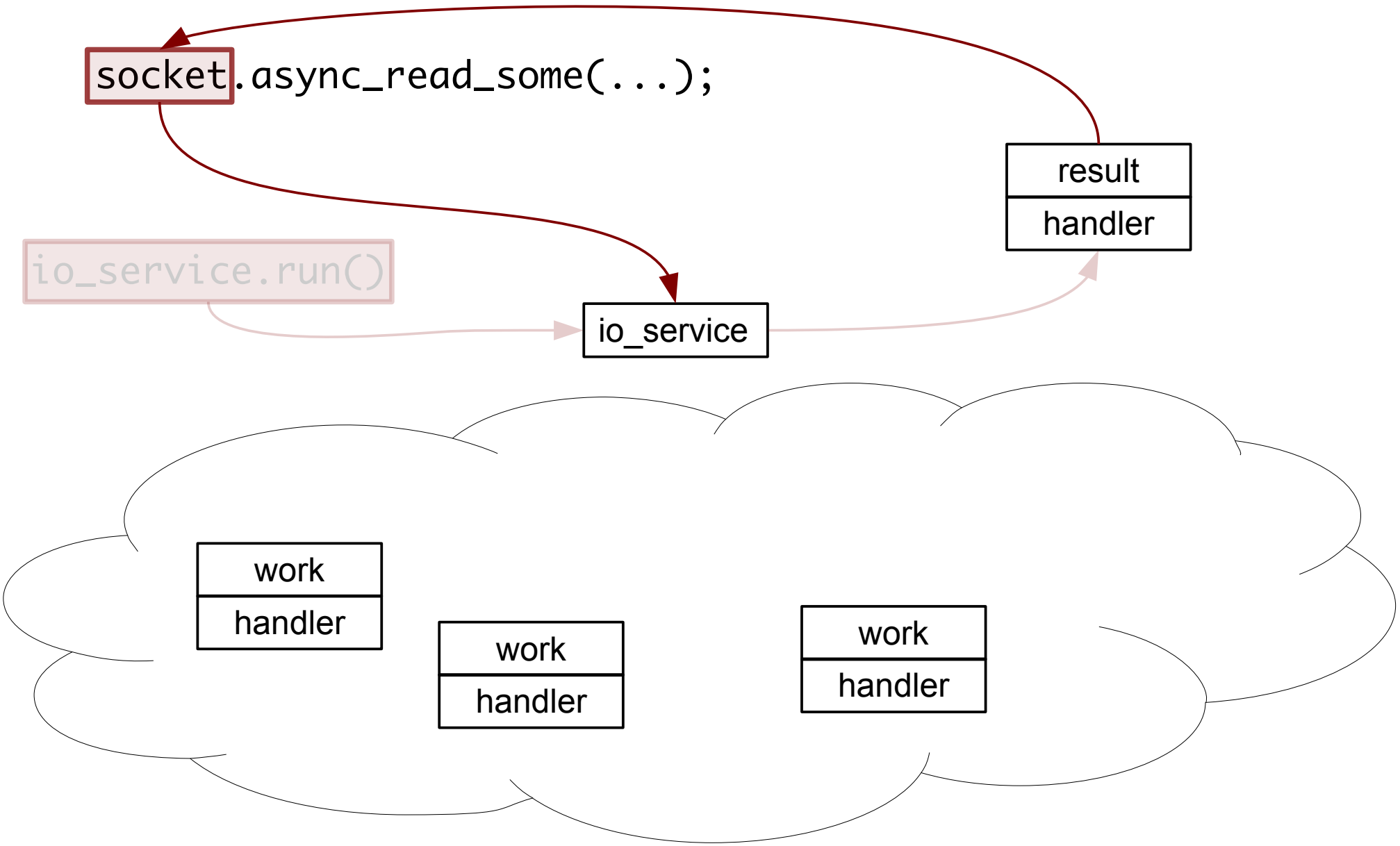
The Basics


io_service.run()
io_service
result
handler
work
handler
work
handler
work
handler

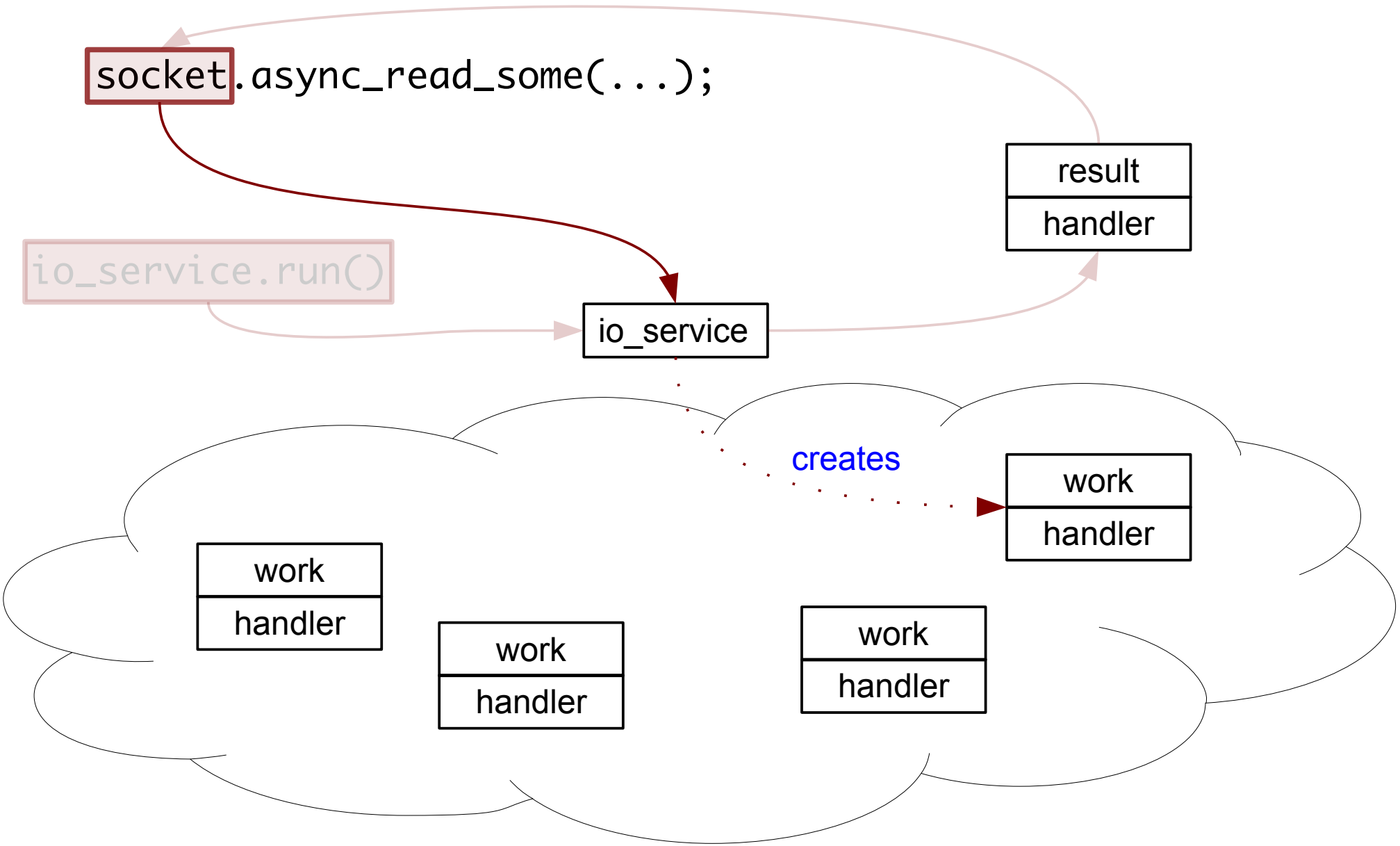
The Basics


socket.async_read_some(...);
result
handler
io_service.run()
io_service
work
handler
work
handler
work
handler

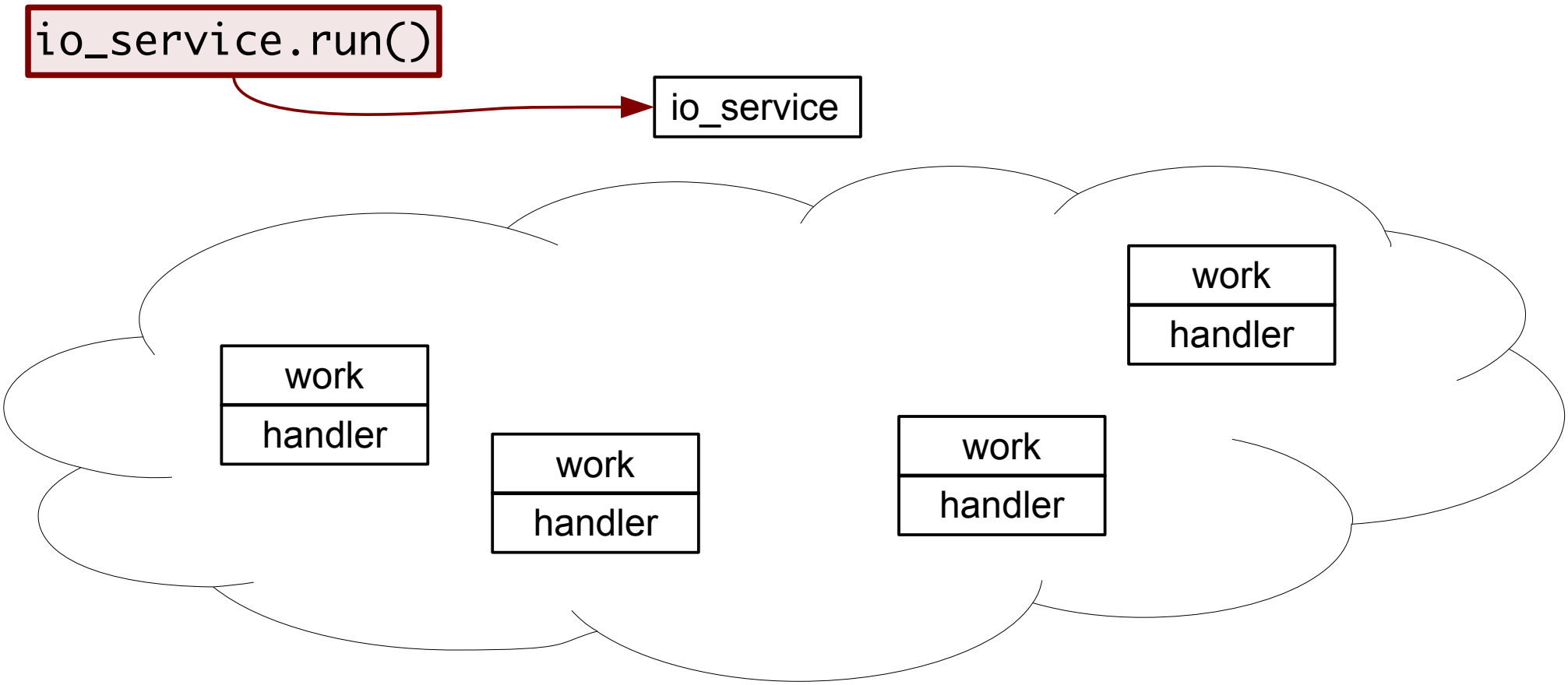
The Basics


socket.async_read_some(...);
io_service.run()
io_service
result
handler
work
handler
work
handler
work
handler

The Basics


socket.async_read_some(...);
io_service.run()
io_service
result
handler
creates
work
handler
work
handler
work
handler
work
handler

The Basics


io_service.run()
io_service
work
handler
work
handler
work
handler
work
handler

The Basics

```
socket.async_connect(
        server_endpoint,
        your_completion_handler);
```

Asynchronous operation

Completion handler

The Basics

```
socket.async_connect(...);
```

```
if (!ec)
{
    socket.async_read_some(...);
}
```

```
if (!ec)
{
    async_write(socket, ...);
}
```

Challenges:

- Object lifetimes
- Thinking asynchronously
- Threads
- Managing complexity

Object Lifetimes

Object Lifetimes

```
socket.async_connect(server_endpoint,
        your_completion_handler_1);

socket.async_read_some(buffers,
        your_completion_handler_2);

async_write(socket, buffers,
        your_completion_handler_3);

acceptor.async_accept(socket,
        peer_endpoint,
        your_completion_handler_4);
```

Object Lifetimes

```
socket.async_connect(server_endpoint,
        your_completion_handler_1);

socket.async_read_some(buffers,
        your_completion_handler_2);

async_write(socket, buffers,
        your_completion_handler_3);

acceptor.async_accept(socket,
        peer_endpoint,
        your_completion_handler_4);
```

By value

Object Lifetimes

```
socket.async_connect(server_endpoint,
        your_completion_handler_1);

socket.async_read_some(buffers,
        your_completion_handler_2);

async_write(socket, buffers,
        your_completion_handler_3);

acceptor.async_accept(socket,
        peer_endpoint,
        your_completion_handler_4);
```

By const reference

Object Lifetimes

```
socket.async_connect(server_endpoint,
        your_completion_handler_1);

socket.async_read_some(buffers,
        your_completion_handler_2);

async_write(socket, buffers,
        your_completion_handler_3);

acceptor.async_accept(socket,
        peer_endpoint,
        your_completion_handler_4);
```

By non-const reference

Object Lifetimes

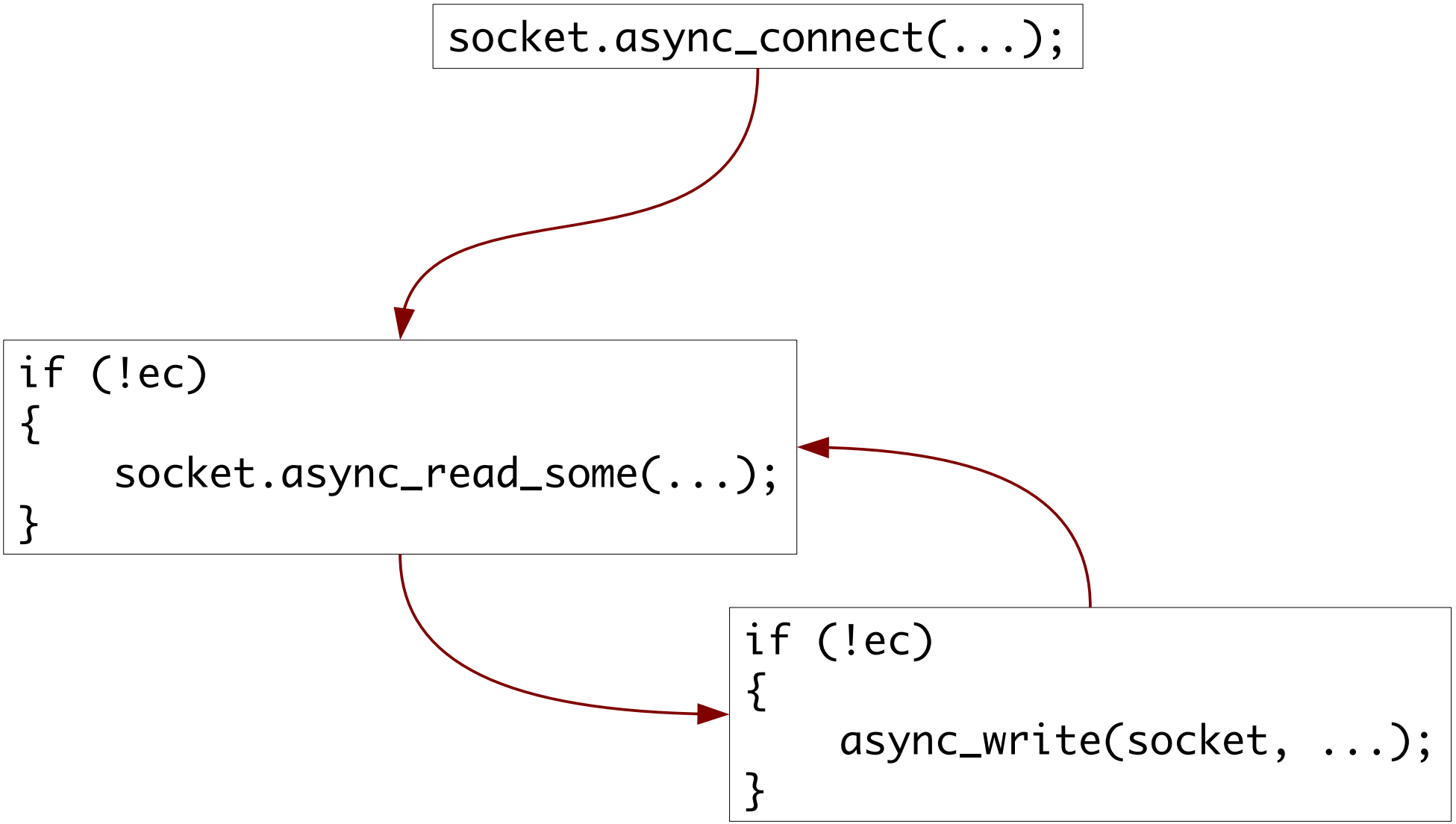
```
socket.async_connect(server_endpoint,
        your_completion_handler_1);

socket.async_read_some(buffers,
        your_completion_handler_2);

async_write(socket, buffers,
        your_completion_handler_3);

acceptor.async_accept(socket,
        peer_endpoint,
        your_completion_handler_4);
```

this pointer

What's wrong with this code?

```
void do_write()
{
    std::string message = ...;
    async_write(socket,
        asio::buffer(message),
        your_completion_handler);
}
```

Object Lifetimes



Object Lifetimes

```
int main
{
    asio::io_service io_service;

    connection conn(io_service);

    io_service.run();
}
```

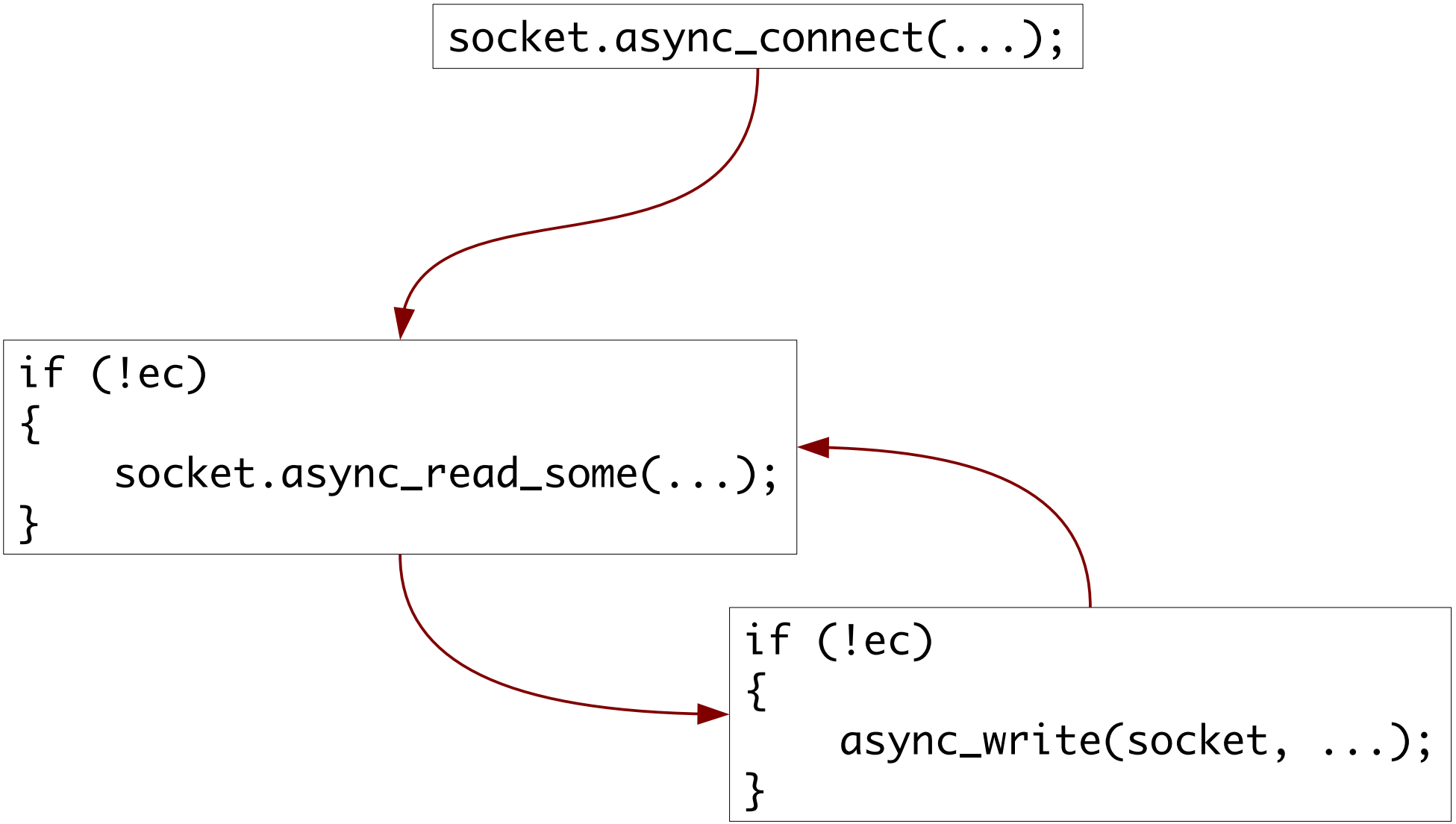
Question:

- When does object lifetime begin and end?

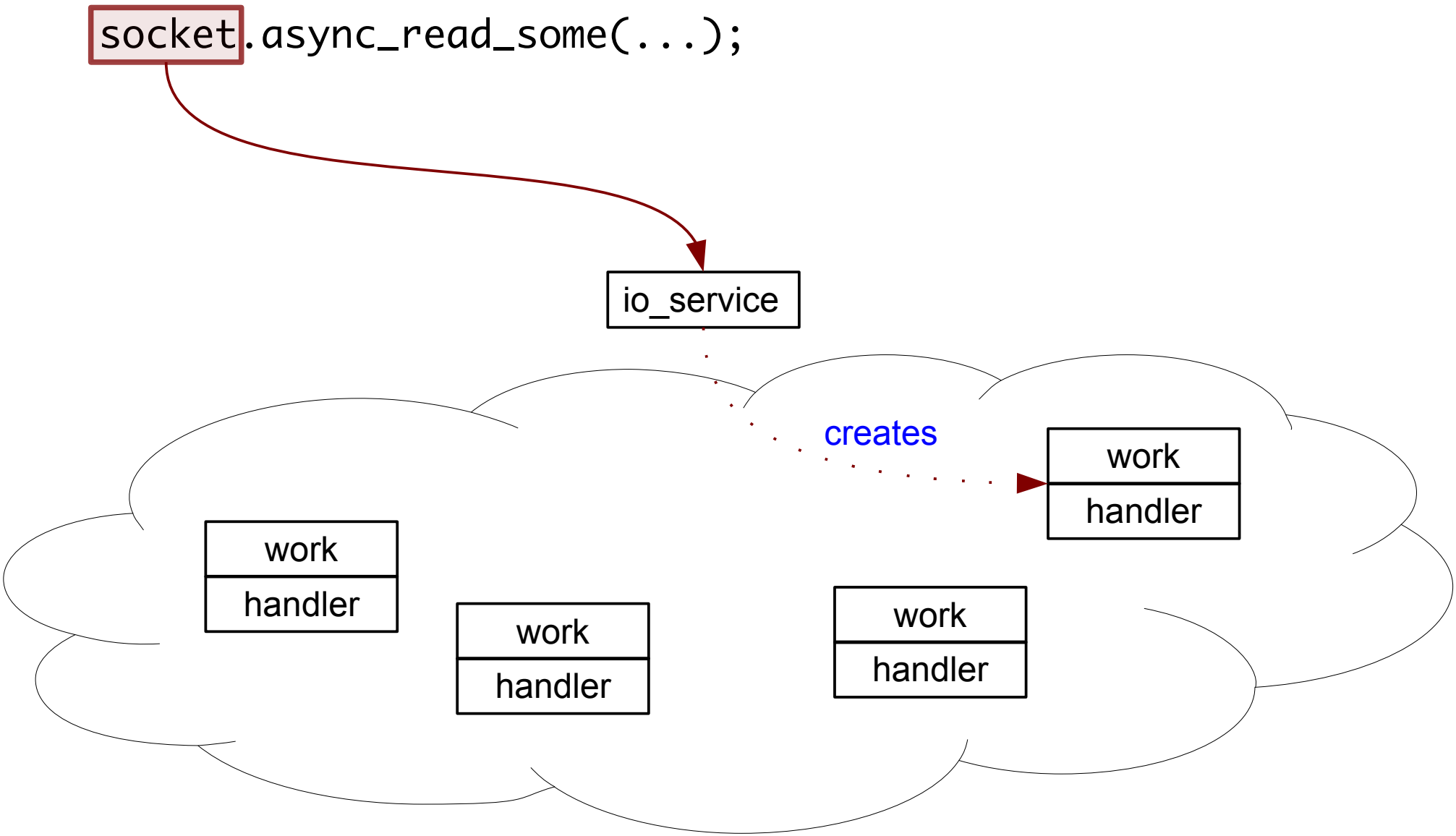
What's wrong with this code?

```
class connection
{
    tcp::socket socket_;
    vector<unsigned char> data_;
    // ...
    ~connection()
    {
        socket_.close();
    }
    // ...
};
```

Object Lifetimes



Object Lifetimes



Object Lifetimes

```
socket.async_connect(server_endpoint,
        your_completion_handler_1);

socket.async_read_some(buffers,
        your_completion_handler_2);

async_write(socket, buffers,
        your_completion_handler_3);

acceptor.async_accept(socket,
        peer_endpoint,
        your_completion_handler_4);
```

By value

Object Lifetimes

```
class connection :
    enable_shared_from_this<connection>
{
    tcp::socket socket_;
    vector<unsigned char> data_;
    // ...
};
```

Object Lifetimes

```
class connection :
    enable_shared_from_this<connection>
{
    tcp::socket socket_;
    vector<unsigned char> data_;
    // ...
    void start_write()
    {
        async_write(socket_,
            asio::buffer(data_),
            bind(&connection::handle_write,
                shared_from_this(), _1, _2));
    }
    // ...
};
```

Object Lifetimes

```
class connection :
    enable_shared_from_this<connection>
{
    tcp::socket socket_;
    vector<unsigned char> data_;
    // ...
    void stop()
    {
        socket_.close();
    }
    // ...
};
```

Object Lifetimes

```
class connection :
    enable_shared_from_this<connection>
{
    tcp::socket socket_;
    vector<unsigned char> data_;
    // ...
    void start()
    {
        socket_.async_connect(...);
    }
    // ...
};
```

Object Lifetimes

```
class connection :
    enable_shared_from_this<connection>
{
    tcp::socket socket_;
    vector<unsigned char> data_;
    // ...
    void start()
    {
        socket_.async_connect(...);
    }
    // ...
};

// ...

make_shared<connection>(...)->start();
```

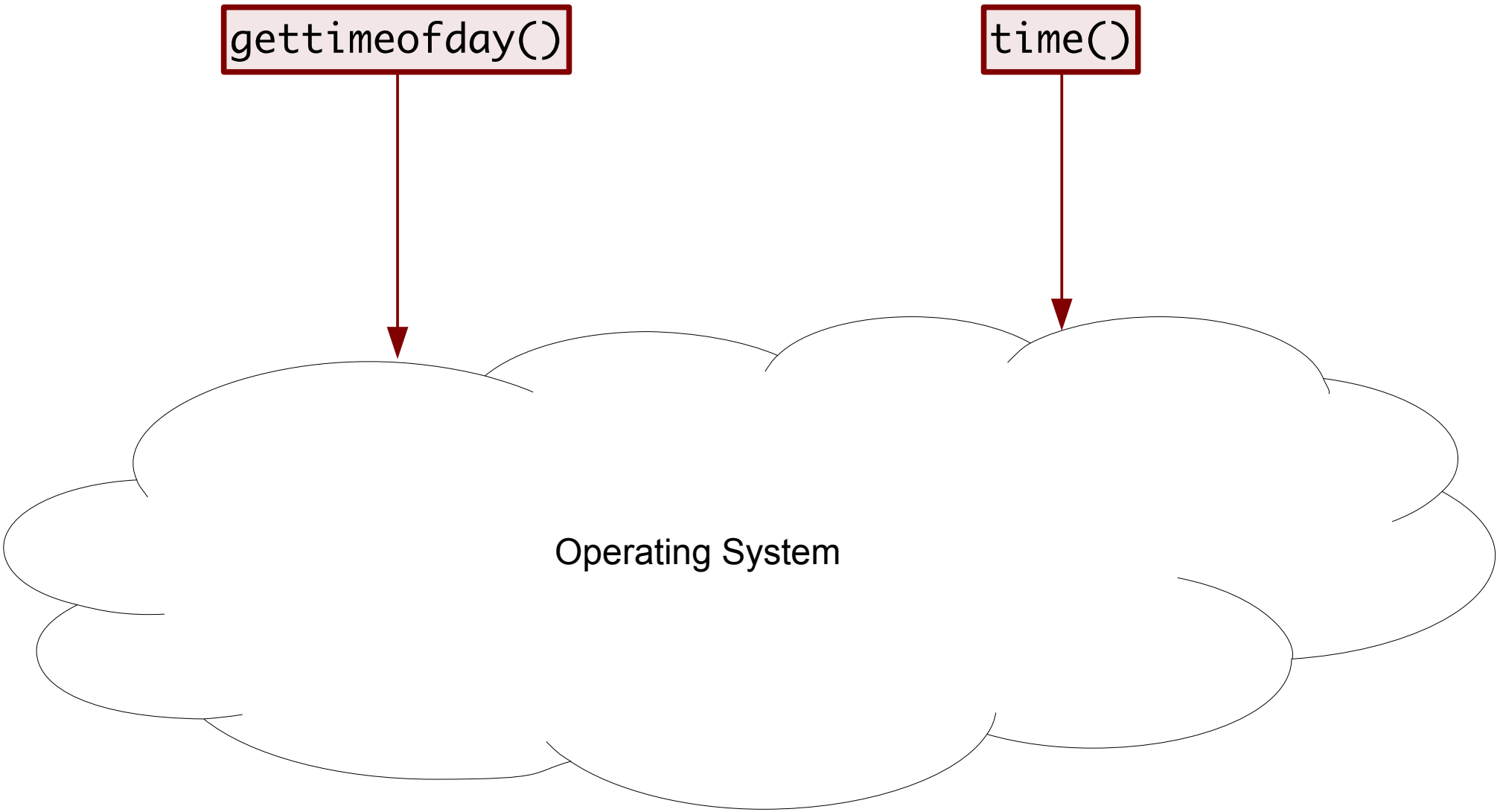
Thinking Asynchronously

```
class connection
{
    tcp::socket socket_;
    vector<unsigned char> data_;
    // ...
    ~connection()
    {
        socket_.close();
    }
    // ...
};
```

What's wrong with this code?

```
double Now()
{
    timeval tv;
    gettimeofday(&tv, 0);
    return time(0) + tv.tv_usec / 1000000.0;
}
```

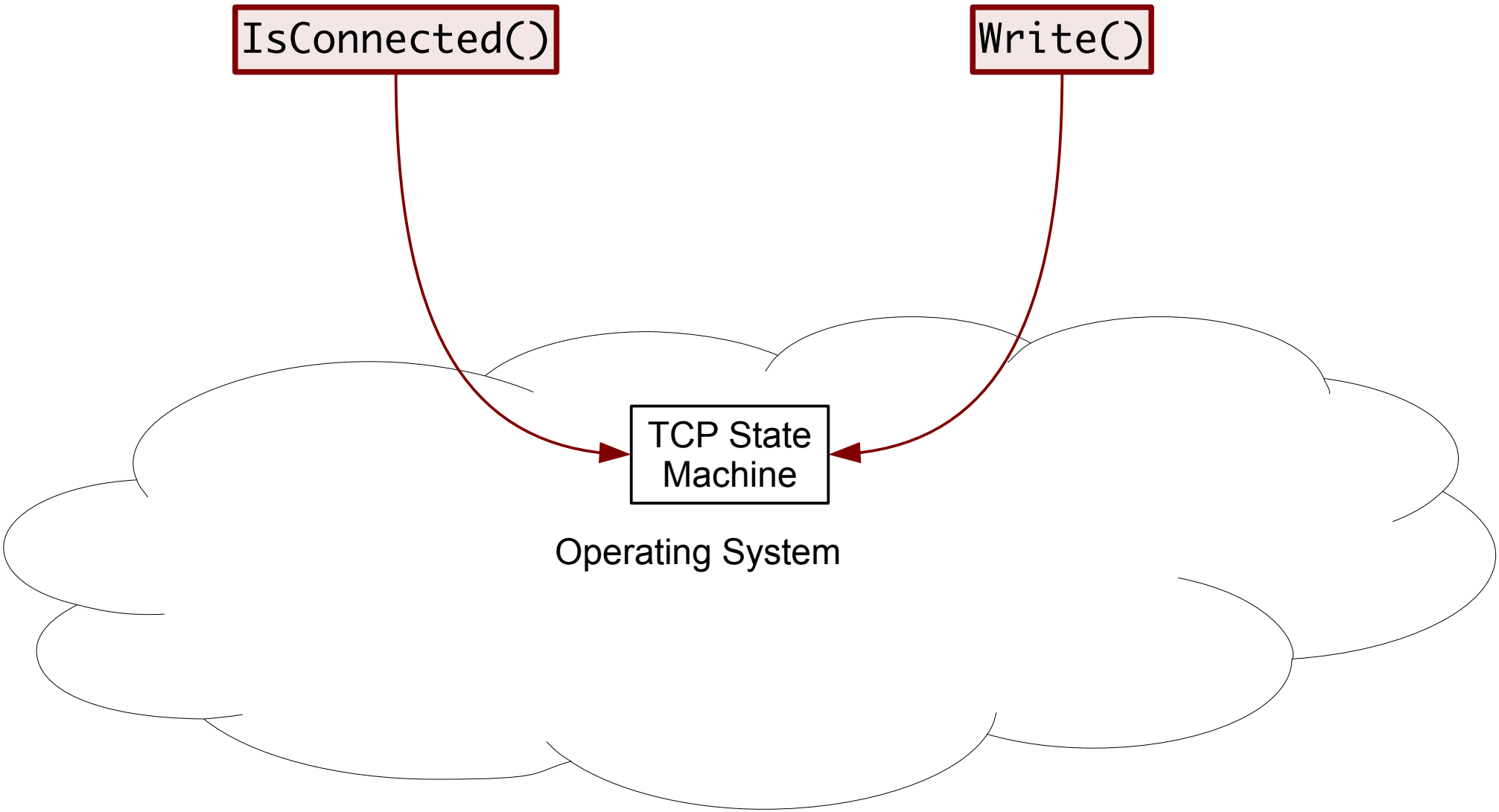
Thinking Asynchronously



What's wrong with this code?

```
if (socket.IsConnected())
{
    socket.Write(data);
}
```

Thinking Asynchronously



```
class connection :
    enable_shared_from_this<connection>
{
    tcp::socket socket_;
    vector<unsigned char> data_;
    // ...
    void stop()
    {
        socket_.close();
    }
    // ...
};
```

```
class connection :
    enable_shared_from_this<connection>
{
    tcp::socket socket_;
    vector<unsigned char> data_;
    // ...
    void handle_write(error_code ec)
    {
        if (!socket_.is_open()) return;
        // ...
    }
    // ...
};
```

```
class connection :
    enable_shared_from_this<connection>
{
    tcp::socket socket_;
    vector<unsigned char> data_;
    // ...
    bool is_stopped() const
    {
        return !socket_.is_open();
    }
    // ...
};
```

```
class connection :
    enable_shared_from_this<connection>
{
    // ...
    void start();
    void stop();
    bool is_stopped() const;
    // ...
};
```

Threads

Spectrum of approaches:

- Single-threaded
- Use threads for long-running tasks
- Multiple io_services, one thread each
- One io_service, many threads

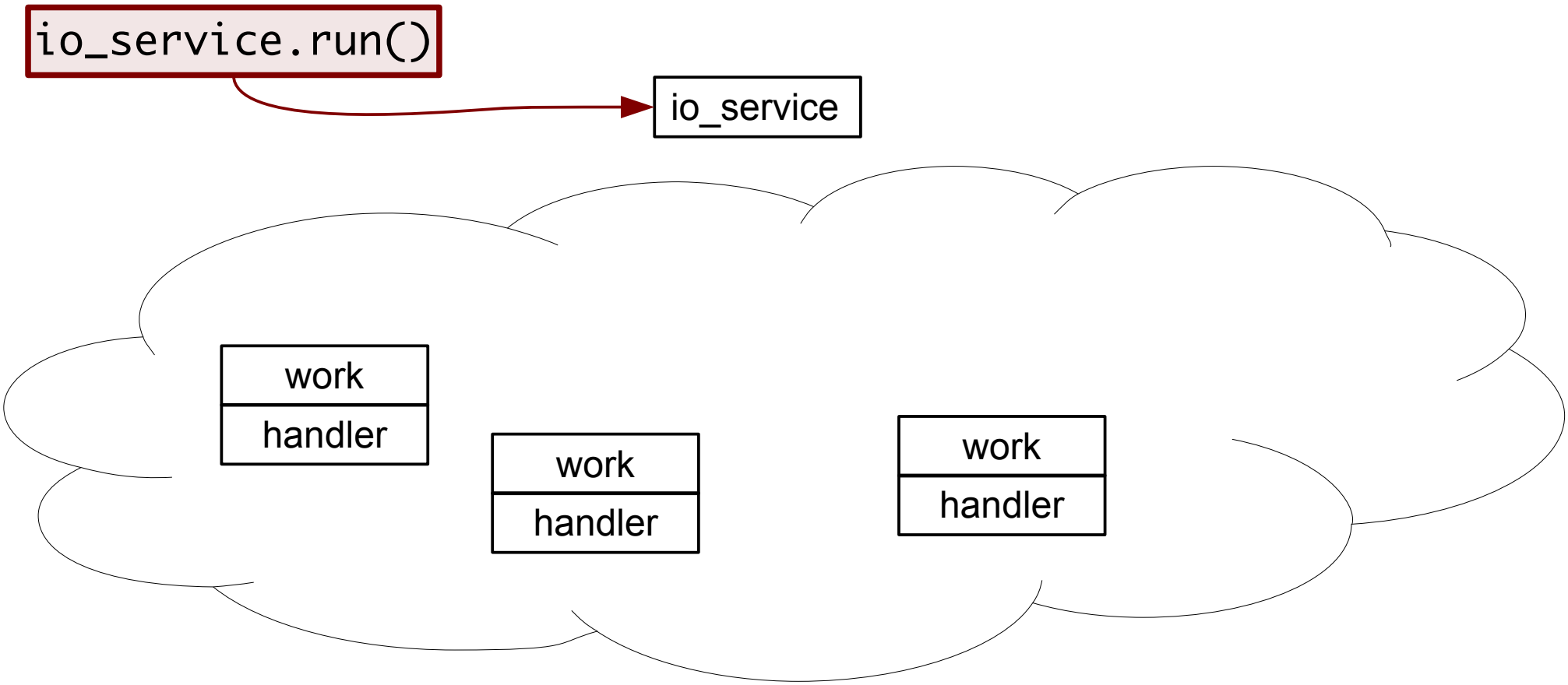
Single-threaded:

- Preferred starting point
- Keep handlers short and non-blocking

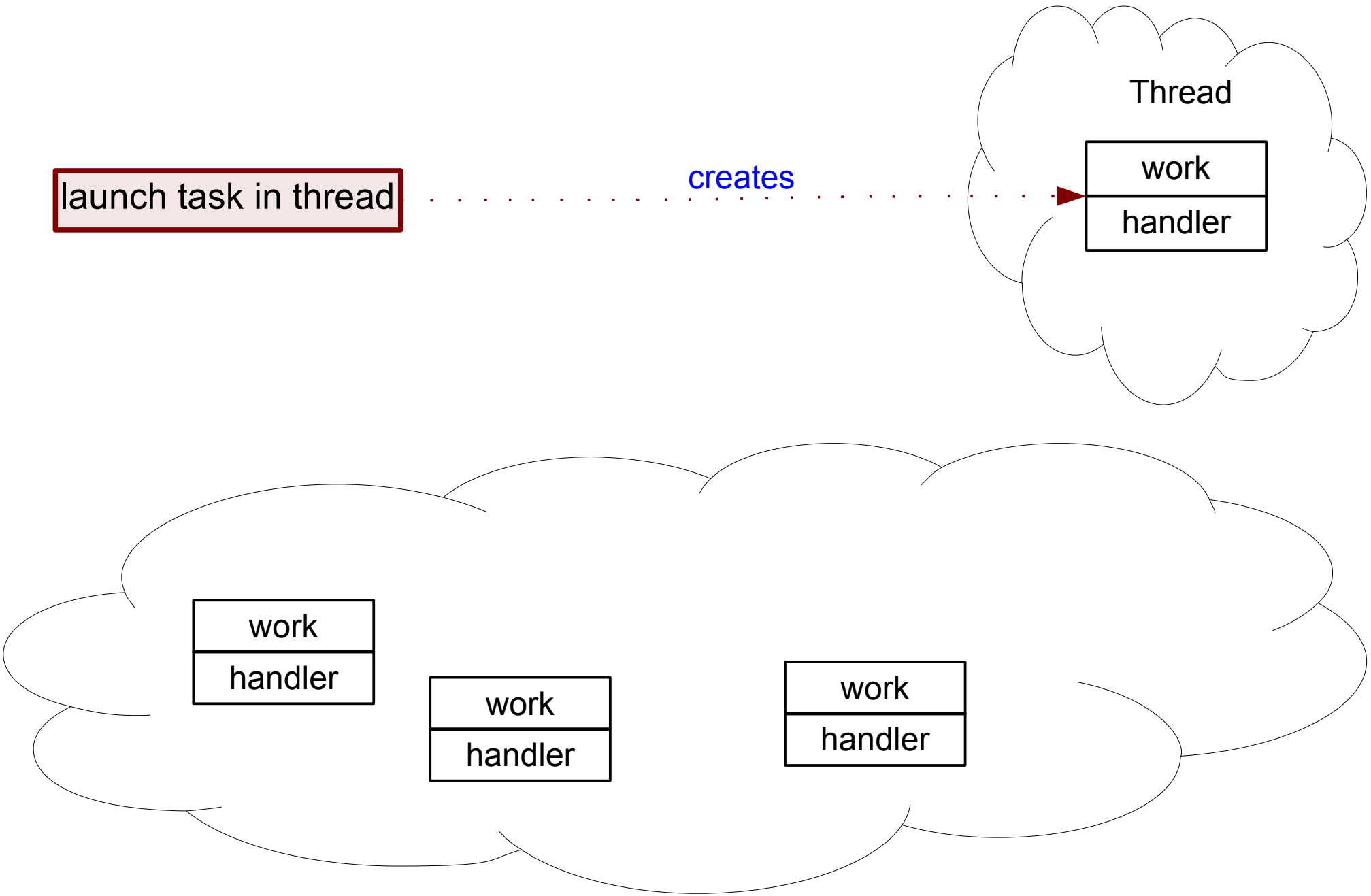
Use threads for long running tasks:

- Logic stays in main thread
- Pass work to background thread
- Pass result back to main thread

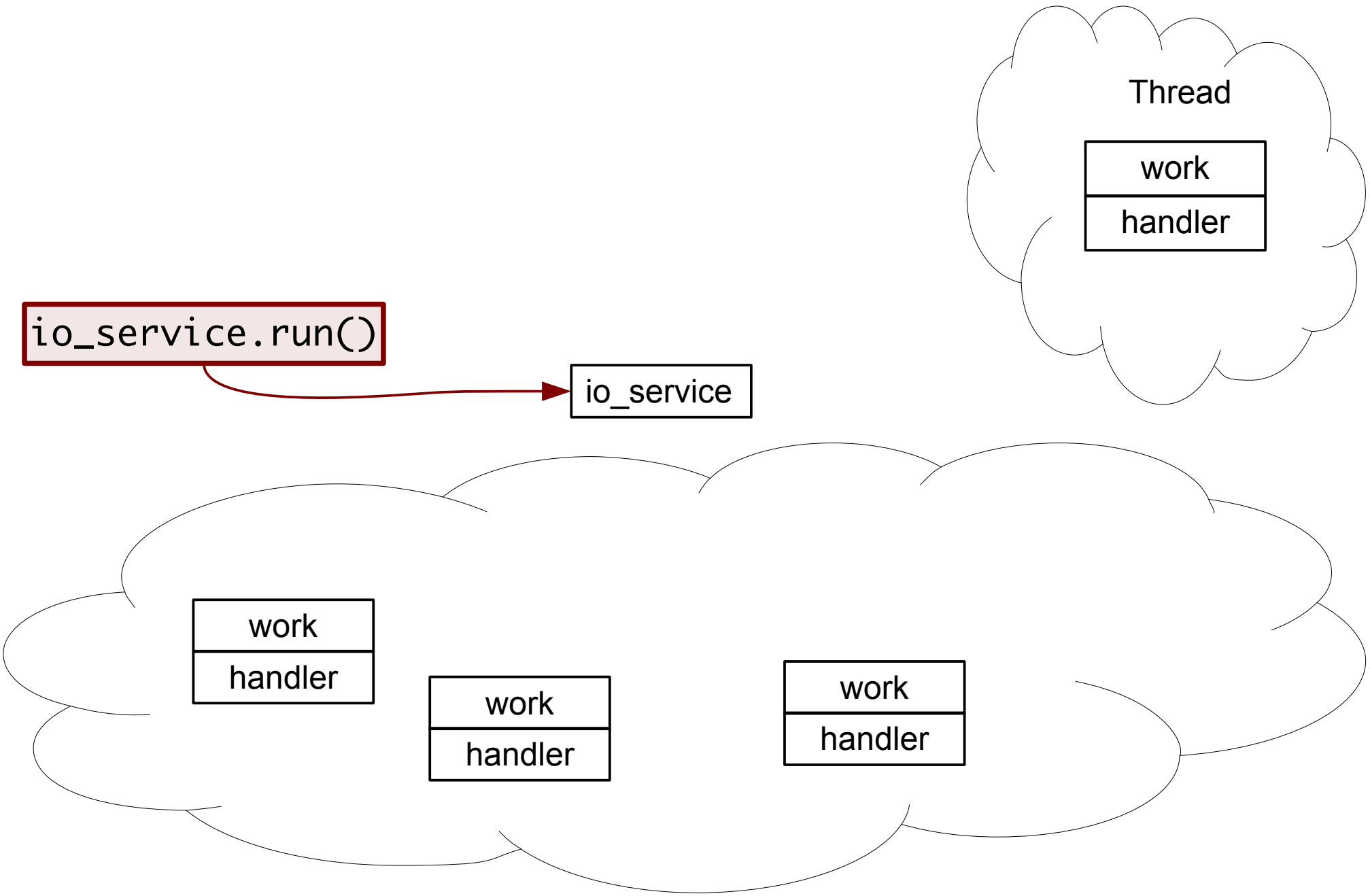
Threads


io_service.run()
io_service
work
handler
work
handler
work
handler

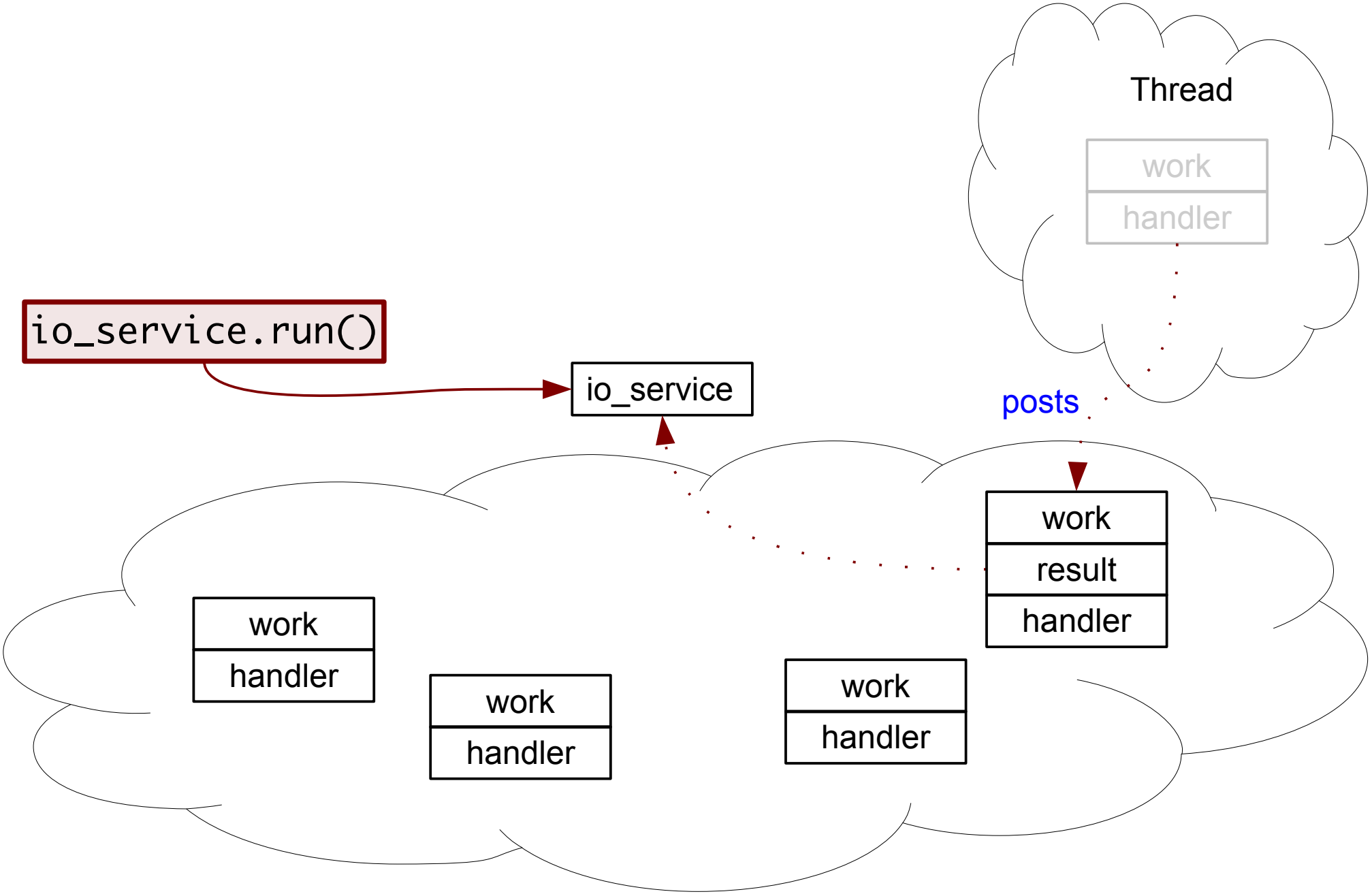
Threads


Thread
work
handler
launch task in thread
creates
work
handler
work
handler
work
handler

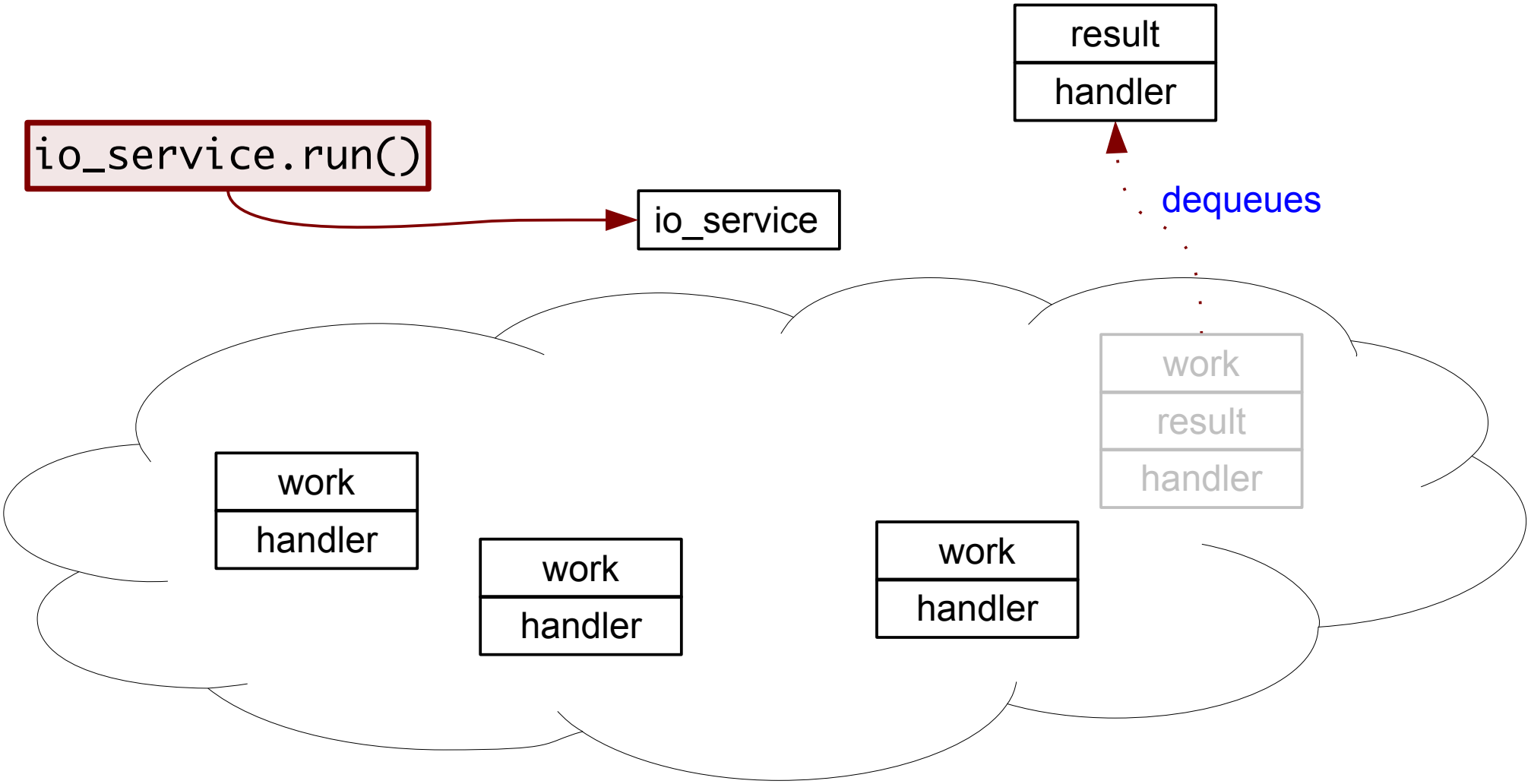
Threads


Thread
work
handler
io_service.run()
io_service
work
handler
work
handler
work
handler

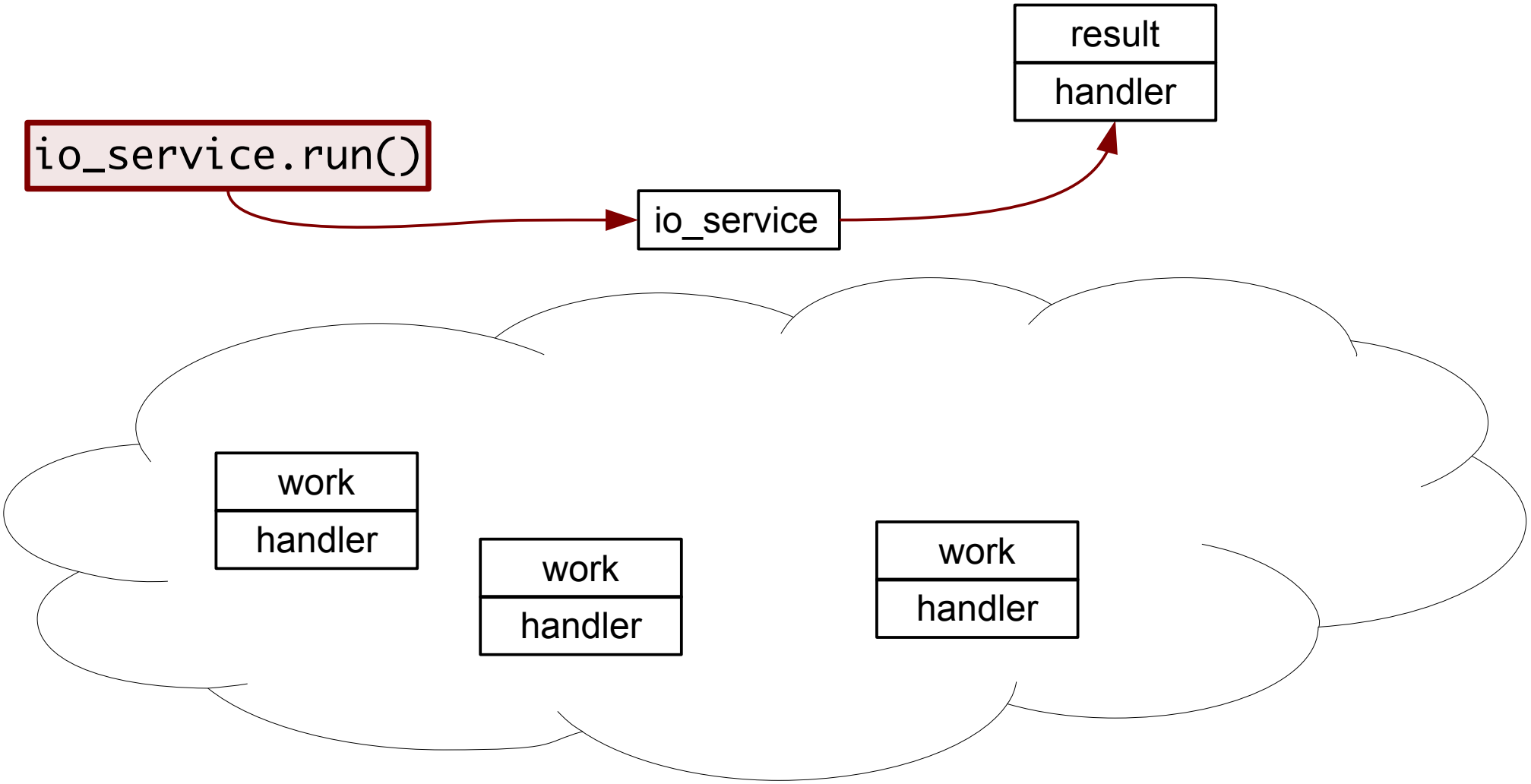
Threads

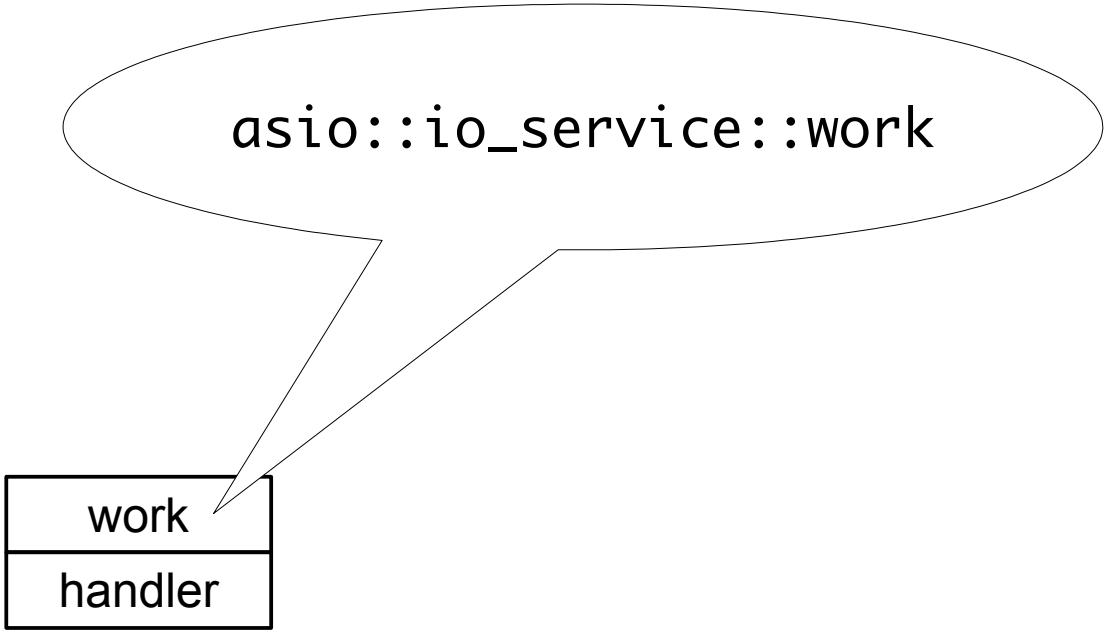

Thread
work
handler
io_service.run()
io_service
posts
work
result
handler
work
handler
work
handler
work
handler

Threads


io_service.run()
io_service
result
handler
dequeues
work
result
handler
work
handler
work
handler
work
handler

Threads


result
handler
io_service.run()
io_service
work
handler
work
handler
work
handler


asio::io_service::work
work
handler

```
class connection :
    enable_shared_from_this<connection>
{
    // ...
    asio::io_service& io_service_;
    // ...
    void start_work()
    {
        async(
            bind(&connection::do_work,
                shared_from_this(),
                ...args...,
                asio::io_service::work(io_service_)));
    }
    // ...
};
```

```
class connection :
    enable_shared_from_this<connection>
{
    // ...
    asio::io_service& io_service_;
    // ...
    void do_work(...args..., asio::io_service::work)
    {
        // long running task ...
        io_service_.post(
            bind(&connection::work_done,
                shared_from_this(),
                ...result...));
    }
    // ...
};
```

```
class connection :
    enable_shared_from_this<connection>
{
    // ...
    void work_done(...result...)
    {
        // ...
    }
    // ...
};
```

Threads

```
asio::io_service async_io_service;
asio::io_service::work async_work(async_io_service);
boost::thread async_thread(
    bind(&asio::io_service::run,
        &async_io_service));

// ...

template <class Handler>
void async(Handler h)
{
    async_io_service.post(h);
}
```

Multiple io_services, one thread each:

- Logic stays in each object's “home” thread
- Keep handlers short and non-blocking
- Objects communicate via “message passing”

```
class connection :
    enable_shared_from_this<connection>
{
    // ...
    asio::io_service& io_service_;
    // ...
    void do_foobar(...args...) { ... }
public:
    void foobar(...args...)
    {
        io_service_.post(
            bind(&connection::do_foobar,
                shared_from_this(),
                ...args...));
    }
    // ...
};
```

```
class connection :
    enable_shared_from_this<connection>
{
    // ...
    asio::io_service& io_service_;
    // ...
    void do_foobar(...args...) { ... }
public:
    void foobar(...args...)
    {
        io_service_.dispatch(
            bind(&connection::do_foobar,
                shared_from_this(),
                ...args...));
    }
    // ...
};
```

Threads

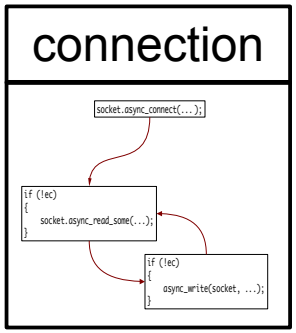
One io_service, multiple threads:

- Handlers can be called on any thread
- Perform logic in strands
- Objects communicate via “message passing”

Strands:

- Non-concurrent invocation of handlers
- May be implicit or explicit

Threads

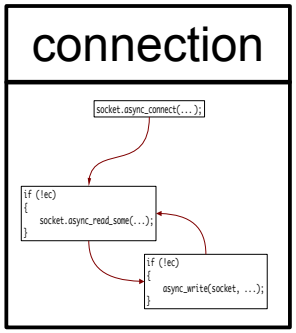


connection

```
socket.async_connect(...);
```

```
if (!ec)
{
    socket.async_read_some(...);
}
```

```
if (!ec)
{
    async_write(socket, ...);
}
```

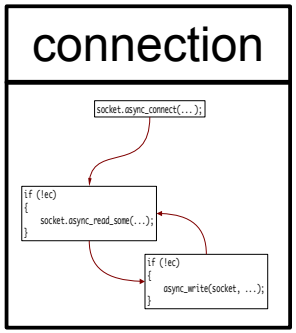


connection

```
socket.async_connect(...);
```

```
if (!ec)
{
    socket.async_read_some(...);
}
```

```
if (!ec)
{
    async_write(socket, ...);
}
```

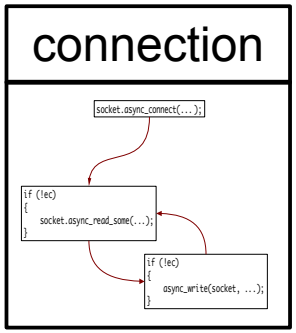


connection

```
socket.async_connect(...);
```

```
if (!ec)
{
    socket.async_read_some(...);
}
```

```
if (!ec)
{
    async_write(socket, ...);
}
```

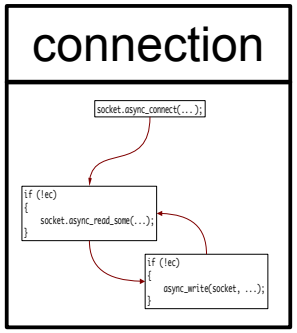


connection

```
socket.async_connect(...);
```

```
if (!ec)
{
    socket.async_read_some(...);
}
```

```
if (!ec)
{
    async_write(socket, ...);
}
```

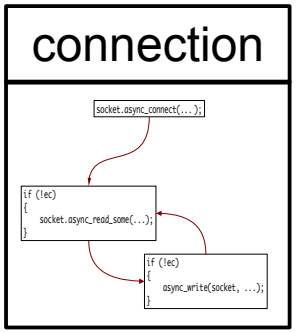


connection

```
socket.async_connect(...);
```

```
if (!ec)
{
    socket.async_read_some(...);
}
```

```
if (!ec)
{
    async_write(socket, ...);
}
```



connection

```
socket.async_connect(...);
```

```
if (!ec)
{
    socket.async_read_some(...);
}
```

```
if (!ec)
{
    async_write(socket, ...);
}
```

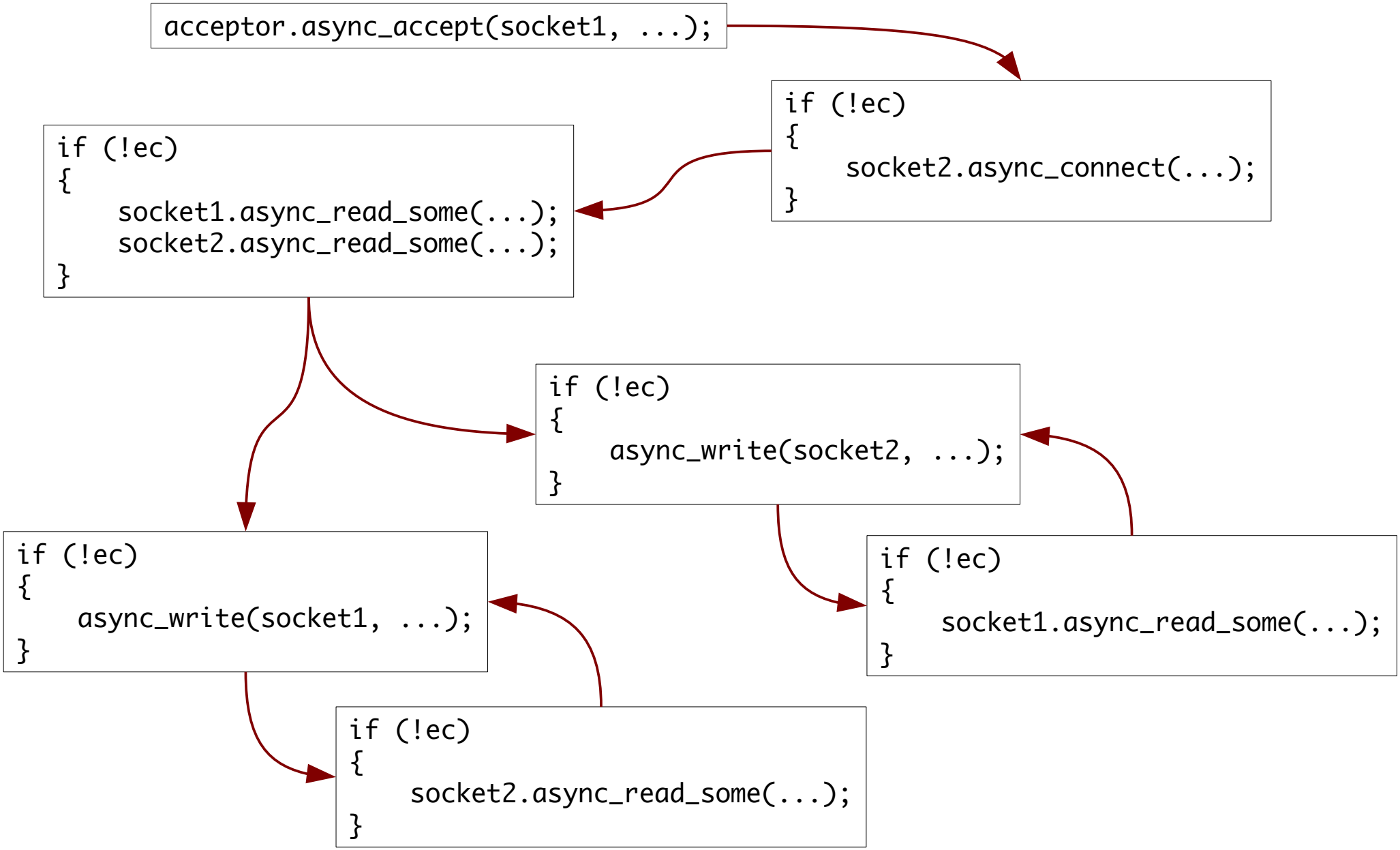
Threads

```
socket.async_connect(...);
```

```
if (!ec)
{
    socket.async_read_some(...);
}
```

```
if (!ec)
{
    async_write(socket, ...);
}
```

Threads

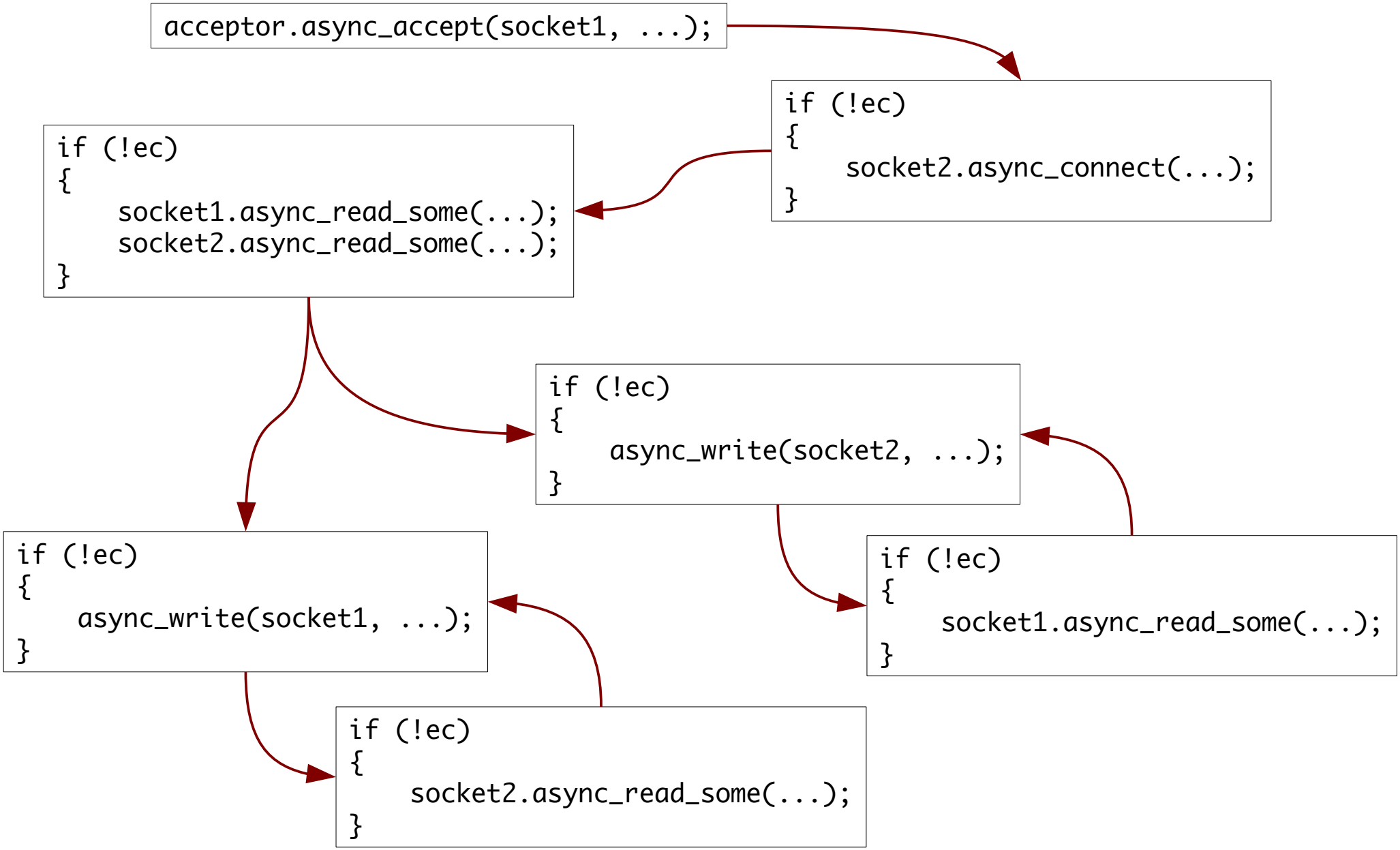

acceptor.async_accept(socket1, ...);
if (!ec)
{
socket2.async_connect(...);
}
if (!ec)
{
socket1.async_read_some(...);
socket2.async_read_some(...);
}
if (!ec)
{
async_write(socket2, ...);
}
if (!ec)
{
socket1.async_read_some(...);
}
if (!ec)
{
async_write(socket1, ...);
}
if (!ec)
{
socket2.async_read_some(...);
}

```
class connection :
    enable_shared_from_this<connection>
{
    tcp::socket socket_;
    vector<unsigned char> data_;
    asio::io_service::strand strand_;
    // ...
    void start_write()
    {
        async_write(socket_,
            asio::buffer(data_),
            strand_.wrap(
                bind(&connection::handle_write,
                    shared_from_this(), _1, _2)));
    }
    // ...
};
```

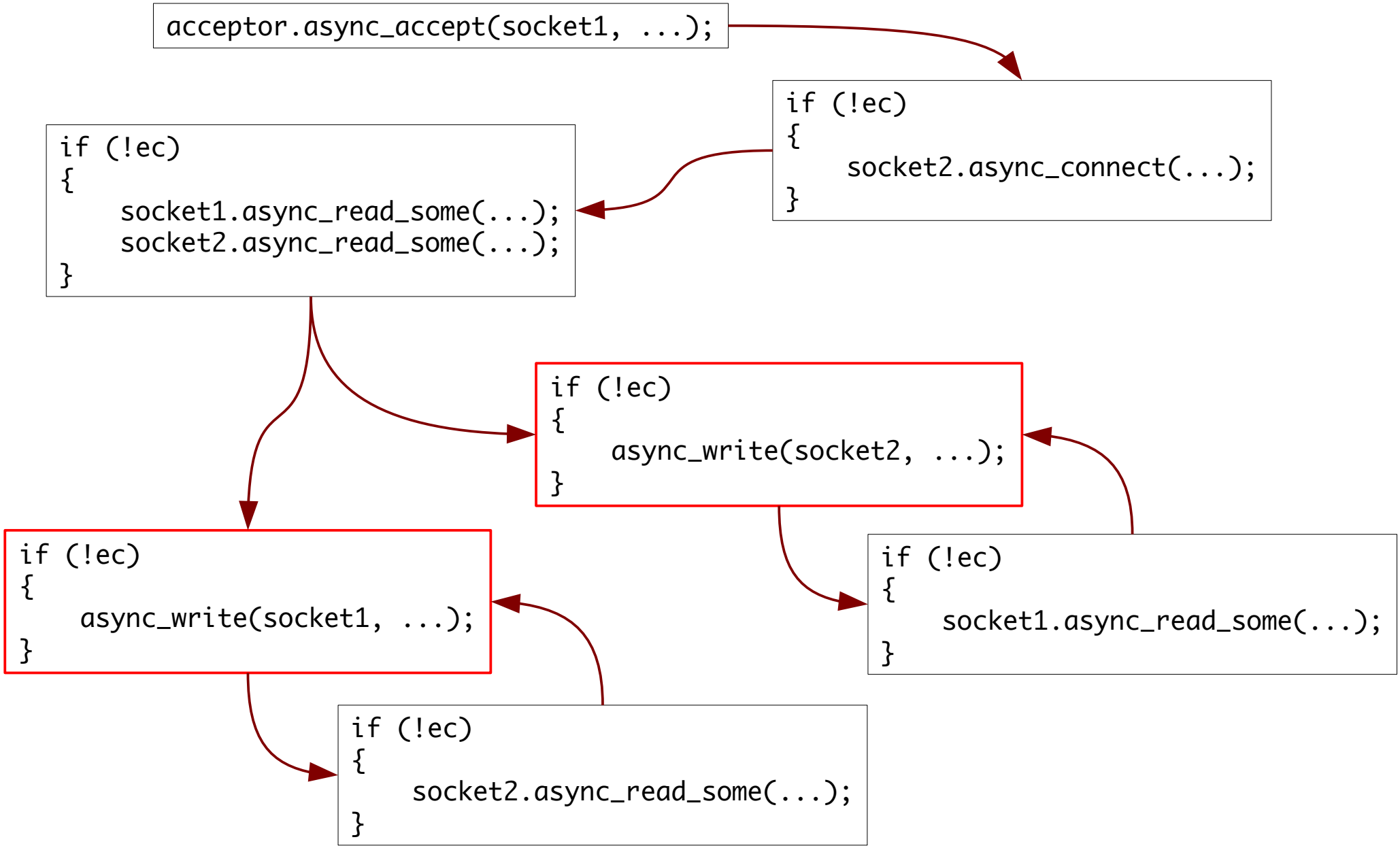
```
class connection :
    enable_shared_from_this<connection>
{
    // ...
    asio::io_service::strand strand_;
    // ...
    void do_foobar(...args...) { ... }
public:
    void foobar(...args...)
    {
        strand_.post(
            bind(&connection::do_foobar,
                shared_from_this(),
                ...args...));
    }
    // ...
};
```

```
class connection :
    enable_shared_from_this<connection>
{
    // ...
    asio::io_service::strand strand_;
    // ...
    void do_foobar(...args...) { ... }
public:
    void foobar(...args...)
    {
        strand_.dispatch(
            bind(&connection::do_foobar,
                shared_from_this(),
                ...args...));
    }
    // ...
};
```

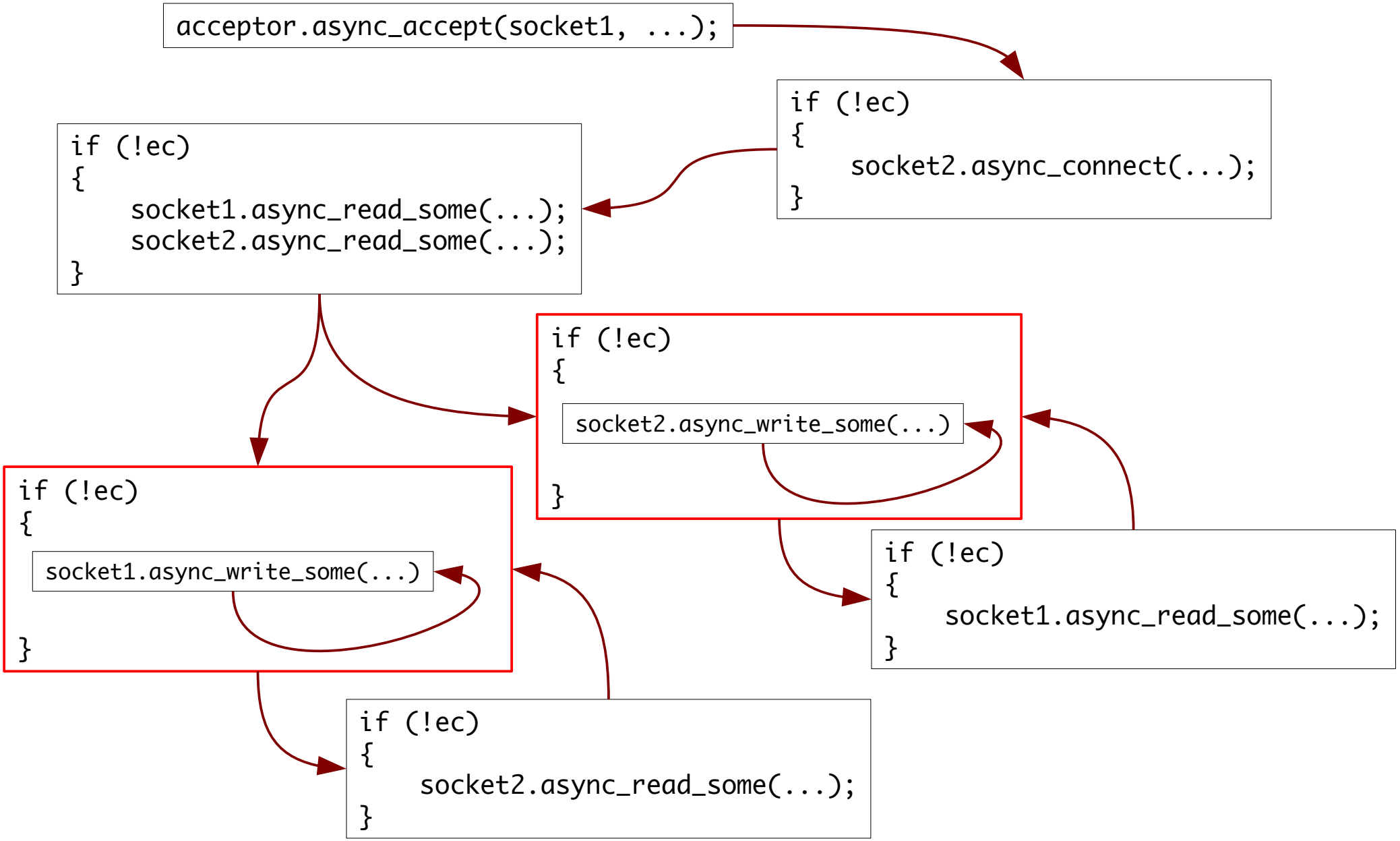
Threads


acceptor.async_accept(socket1, ...);
if (!ec)
{
socket2.async_connect(...);
}
if (!ec)
{
socket1.async_read_some(...);
socket2.async_read_some(...);
}
if (!ec)
{
async_write(socket2, ...);
}
if (!ec)
{
socket1.async_read_some(...);
}
if (!ec)
{
async_write(socket1, ...);
}
if (!ec)
{
socket2.async_read_some(...);
}

Threads


acceptor.async_accept(socket1, ...);
if (!ec)
{
socket2.async_connect(...);
}
if (!ec)
{
socket1.async_read_some(...);
socket2.async_read_some(...);
}
if (!ec)
{
async_write(socket2, ...);
}
if (!ec)
{
socket1.async_read_some(...);
}
if (!ec)
{
async_write(socket1, ...);
}
if (!ec)
{
socket2.async_read_some(...);
}

Threads


acceptor.async_accept(socket1, ...);
if (!ec)
{
socket2.async_connect(...);
}
if (!ec)
{
socket1.async_read_some(...);
socket2.async_read_some(...);
}
if (!ec)
{
socket2.async_write_some(...)
}
if (!ec)
{
socket1.async_write_some(...)
}
if (!ec)
{
socket1.async_read_some(...);
}
if (!ec)
{
socket2.async_read_some(...);
}

```
template <class Handler>
struct mutex_wrapper
{
    mutex& mutex_;
    Handler handler_;

    mutex_wrapper(mutex& m, Handler h) :
        mutex_(m), handler_(h) {}

    void operator()() { handler_(); }

    template <class Arg1>
    void operator()(Arg1 a1) { handler_(a1); }

    template <class Arg1, class Arg2>
    void operator()(Arg1 a1, Arg2 a2) { handler_(a1, a2); }
};
```

```
template <class Handler>
mutex_wrapper<Handler> wrap(mutex& m, Handler h)
{
    return mutex_wrapper<Handler>(m, h);
}
```

```
template <class Function, class Handler>
void asio_handler_invoke(
    Function f,
    mutex_wrapper<Handler>* w)
{
    mutex::scoped_lock lock(w->mutex_);
    f();
}
```

Invocation hook

```
class connection :
    enable_shared_from_this<connection>
{
    tcp::socket socket_;
    vector<unsigned char> data_;
    mutex mutex_;
    // ...
    void start_write()
    {
        async_write(socket_,
            asio::buffer(data_),
            wrap(mutex_,
                bind(&connection::handle_write,
                    shared_from_this(), _1, _2)));
    }
    // ...
};
```

Managing Complexity

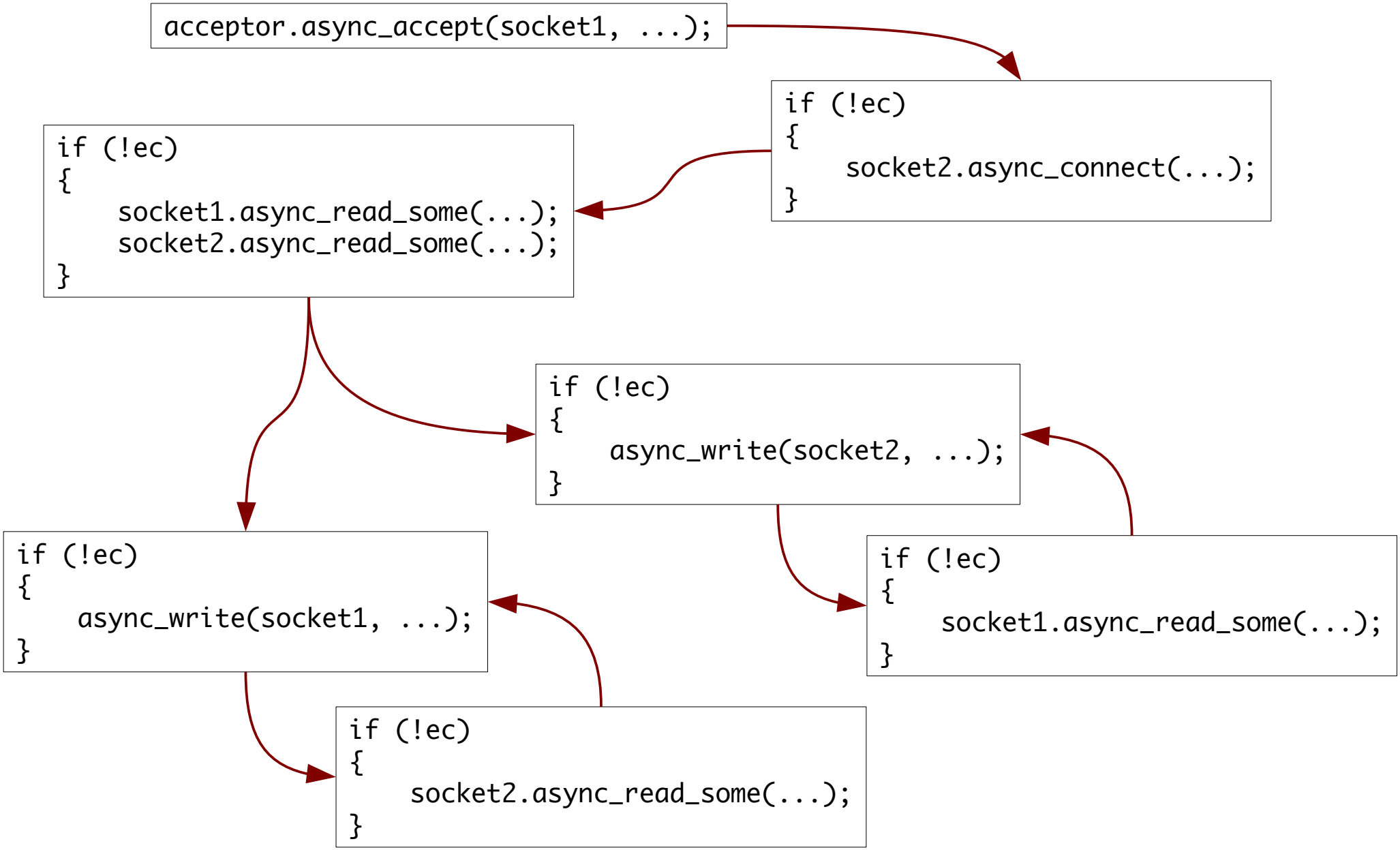
Approaches:

- Pass the buck
- The buck stops here

Approaches:

- Pass the buck
  - Functions
  - Classes
- The buck stops here
  - Classes

Managing Complexity



A “pass the buck” function:

```
template <class Handler>
void async_transfer(
    tcp::socket& socket1, tcp::socket& socket2,
    asio::mutable_buffers_1 working_buffer,
    Handler handler);
```

Also known as a “composed operation”

Managing Complexity

Initiating function

```
if (!ec)
{
    socket1.async_read_some(...);
}
else
{
    handler(ec);
}
```

```
if (!ec)
{
    async_write(socket2, ...);
}
else
{
    handler(ec);
}
```

Managing Complexity

```
template <class Handler>
void do_read(
    tcp::socket& socket1, tcp::socket& socket2,
    asio::mutable_buffers_1 working_buffer,
    tuple<Handler> handler,
    const error_code& ec)
{
    if (!ec)
    {
        socket1.async_read_some(
            working_buffer,
            bind(&do_write<Handler>,
                ref(socket1), ref(socket2),
                working_buffer,
                handler, _1, _2));
    }
    else
    {
        get<0>(handler)(ec);
    }
}
```

Managing Complexity

```
template <class Handler>
void do_write(
    tcp::socket& socket1, tcp::socket& socket2,
    asio::mutable_buffers_1 working_buffer,
    tuple<Handler> handler,
    const error_code& ec, size_t length)
{
    if (!ec)
    {
        asio::async_write(socket2,
            asio::buffer(working_buffer, length),
            bind(&do_read<Handler>,
                ref(socket1), ref(socket2),
                working_buffer,
                handler, _1));
    }
    else
    {
        get<0>(handler)(ec);
    }
}
```

Managing Complexity

```
template <class Handler>
void async_transfer(
    tcp::socket& socket1, tcp::socket& socket2,
    asio::mutable_buffers_1 working_buffer,
    Handler handler)
{
    do_read(
        socket1, socket2,
        working_buffer,
        make_tuple(handler),
        error_code());
}
```

Managing Complexity

Initiating function

```
if (!ec)
{
    if (do_read)
    {
        do_read = false;
        socket1.async_read_some(...);
    }
    else
    {
        do_read = true;
        async_write(socket2, ...);
    }
}
else
{
    handler(ec);
}
```

Managing Complexity

```
template <class Handler>
struct transfer_op
{
    bool do_read;
    tcp::socket& socket1; tcp::socket& socket2;
    asio::mutable_buffer working_buffer;
    Handler handler;
    void operator()(const error_code& ec, size_t length);
};
```

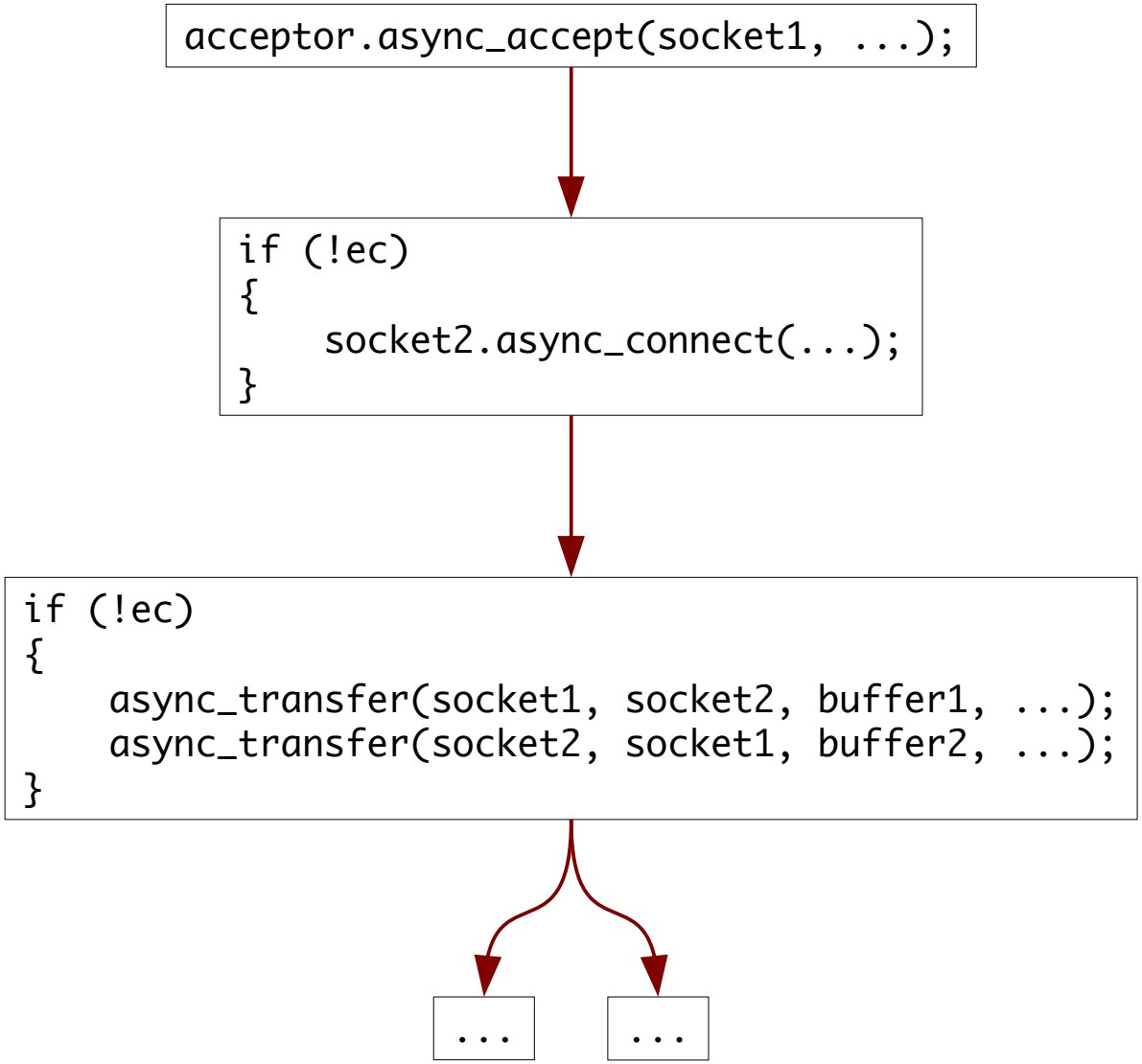
Managing Complexity

```
template <class Handler>
void transfer_op::operator()(const error_code& ec, size_t length)
{
    if (!ec)
    {
        if (do_read)
        {
            do_read = false;
            socket1.async_read_some(working_buffer, *this);
        }
        else
        {
            do_read = true;
            asio::async_write(socket2,
                asio::buffer(working_buffer, length), *this);
        }
    }
    else
    {
        handler(ec);
    }
};
```

Managing Complexity

```
template <class Handler>
void async_transfer(
    tcp::socket& socket1, tcp::socket& socket2,
    asio::mutable_buffers_1 working_buffer,
    Handler handler)
{
    transfer_op<Handler> op = {
        true, socket1, socket2,
        working_buffer, handler };
    op(error_code(), 0);
}
```

Managing Complexity



Managing Complexity

```
template <class Function, class Handler>
void asio_handler_invoke(const Function& f,
    transfer_op<Handler>* op)
{
  using boost::asio::asio_handler_invoke;
  asio_handler_invoke(f, addressof(op->handler));
}
```

Invocation hook

Managing Complexity

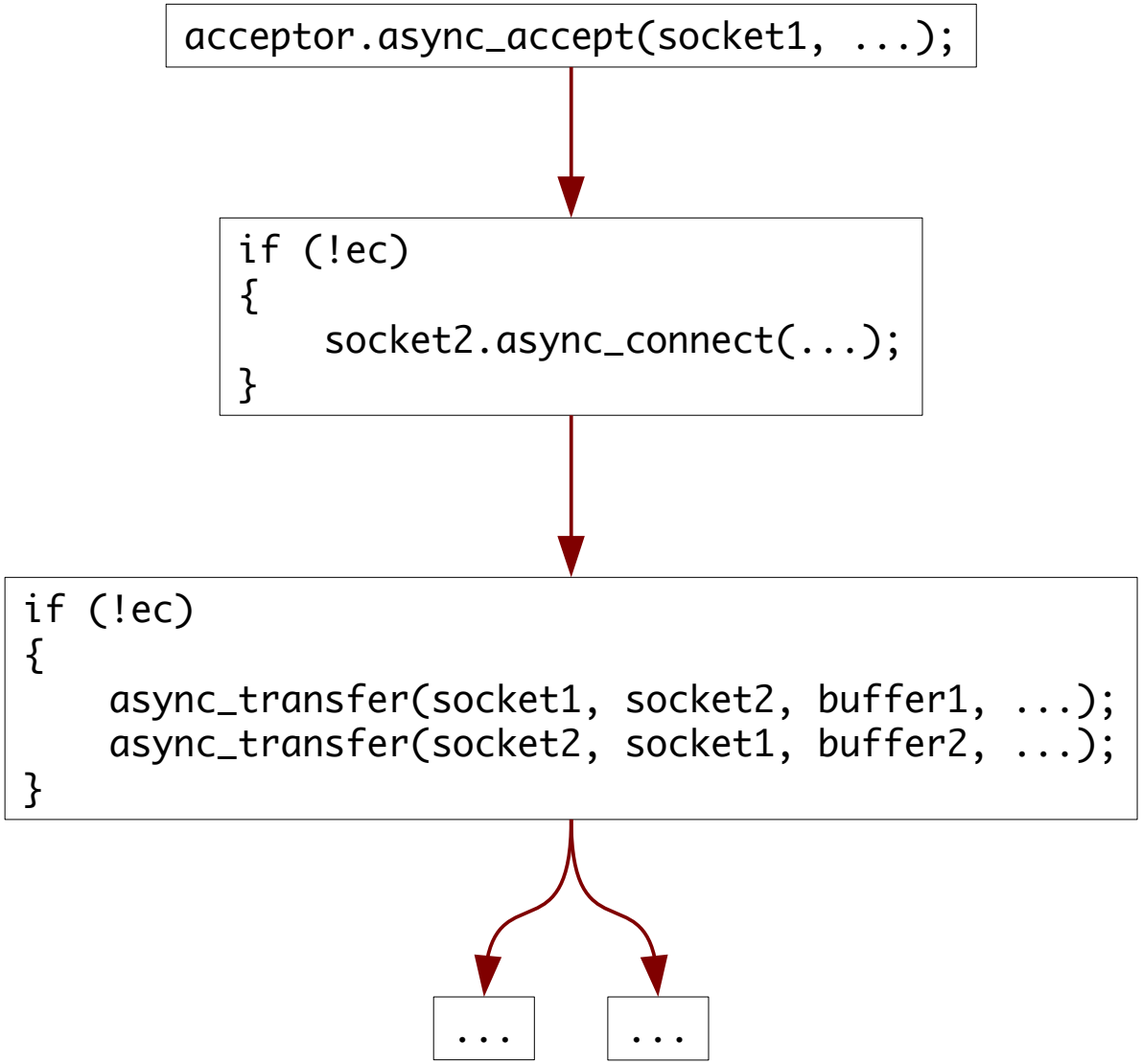
```
template <class Handler>
void* asio_handler_allocate(size_t n,
    transfer_op<Handler>* op)
{
  using boost::asio::asio_handler_allocate;
  return asio_handler_allocate(n, addressof(op->handler));
}

template <class Handler>
void asio_handler_deallocate(void* p, size_t n,
    transfer_op<Handler>* op)
{
  using boost::asio::asio_handler_deallocate;
  asio_handler_deallocate(p, n, addressof(op->handler));
}
```

Allocation hooks

Managing Complexity

```
template <class Stream1, class Stream2, class Handler>
void async_transfer(
        Stream1& stream1, Stream2& stream2,
        asio::mutable_buffers_1 working_buffer,
        Handler handler);
```

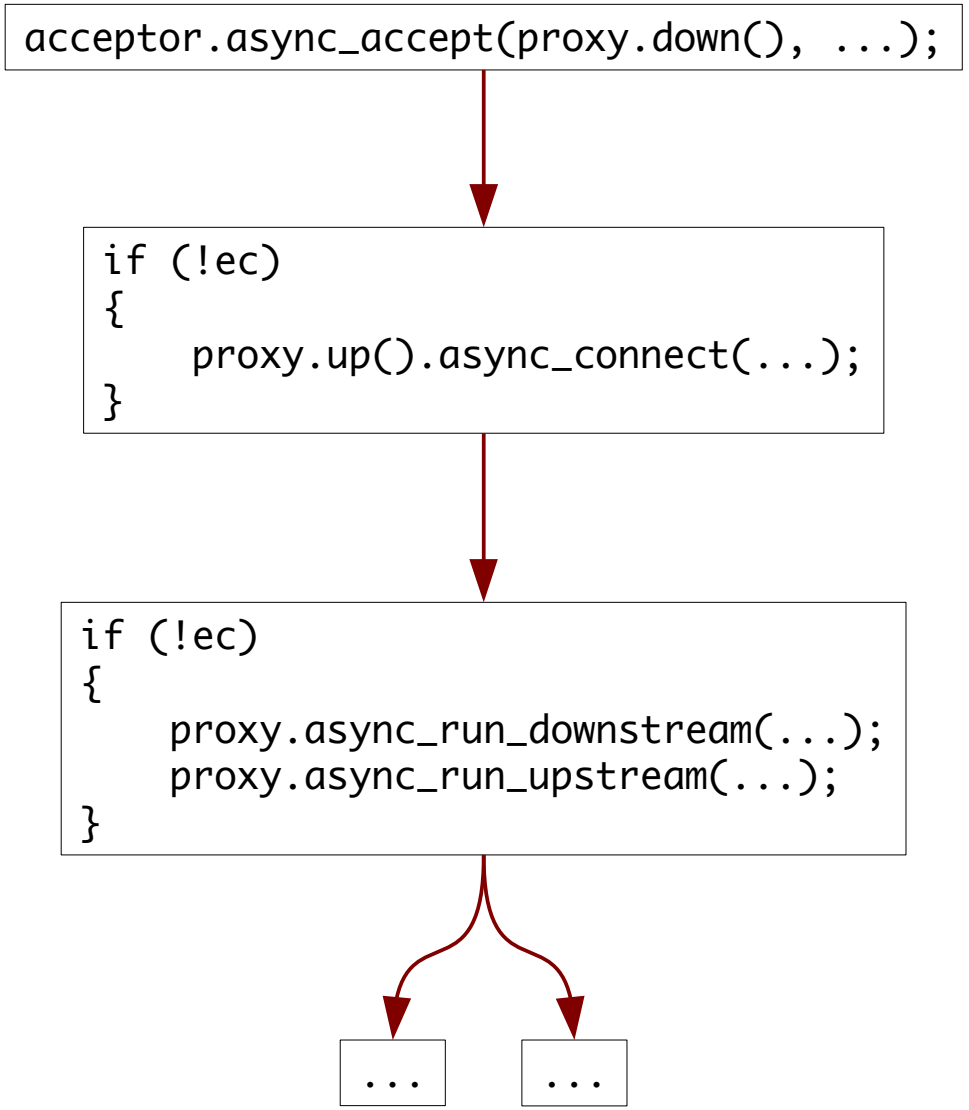
Managing Complexity



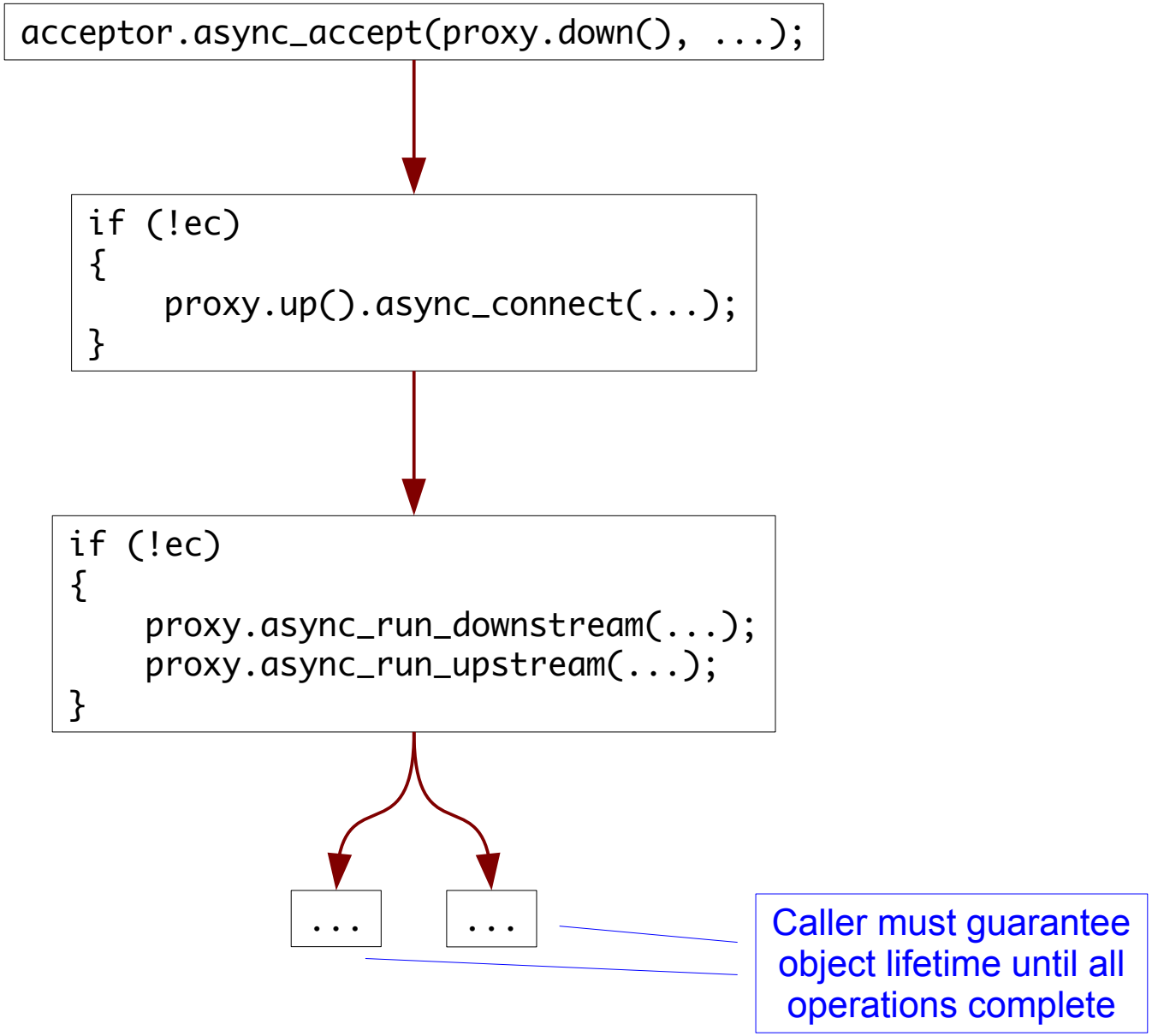
A “pass the buck” class:

```
template <class Stream1, class Stream2>
class proxy
{
    Stream1 up_;
    Stream2 down_;
    vector<unsigned char> buffer1_, buffer2_;

public:
    proxy(...);
    Stream1& up() { return up_; }
    Stream2& down() { return down_; }

    template <class Handler>
    void async_run_upstream(Handler handler);

    template <class Handler>
    void async_run_downstream(Handler handler);
};
```

Managing Complexity



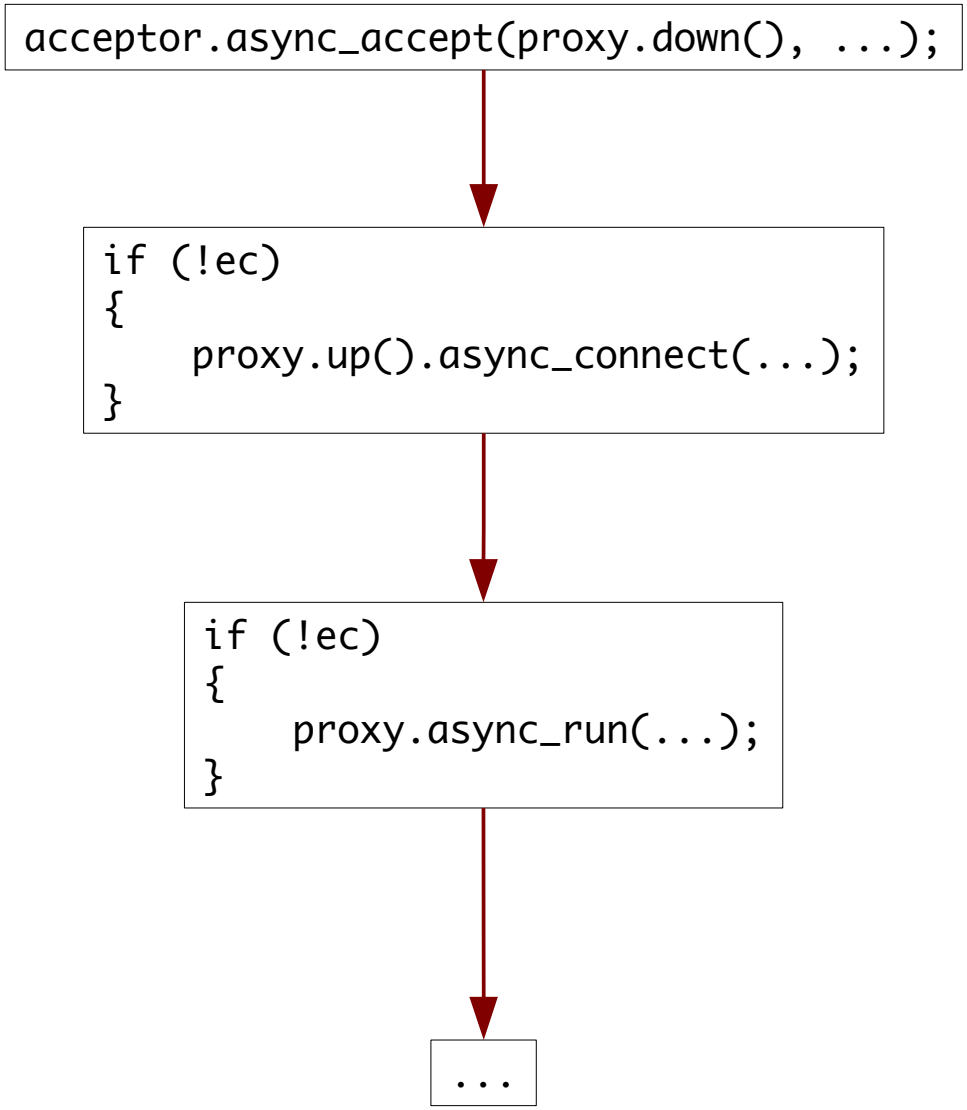
Managing Complexity



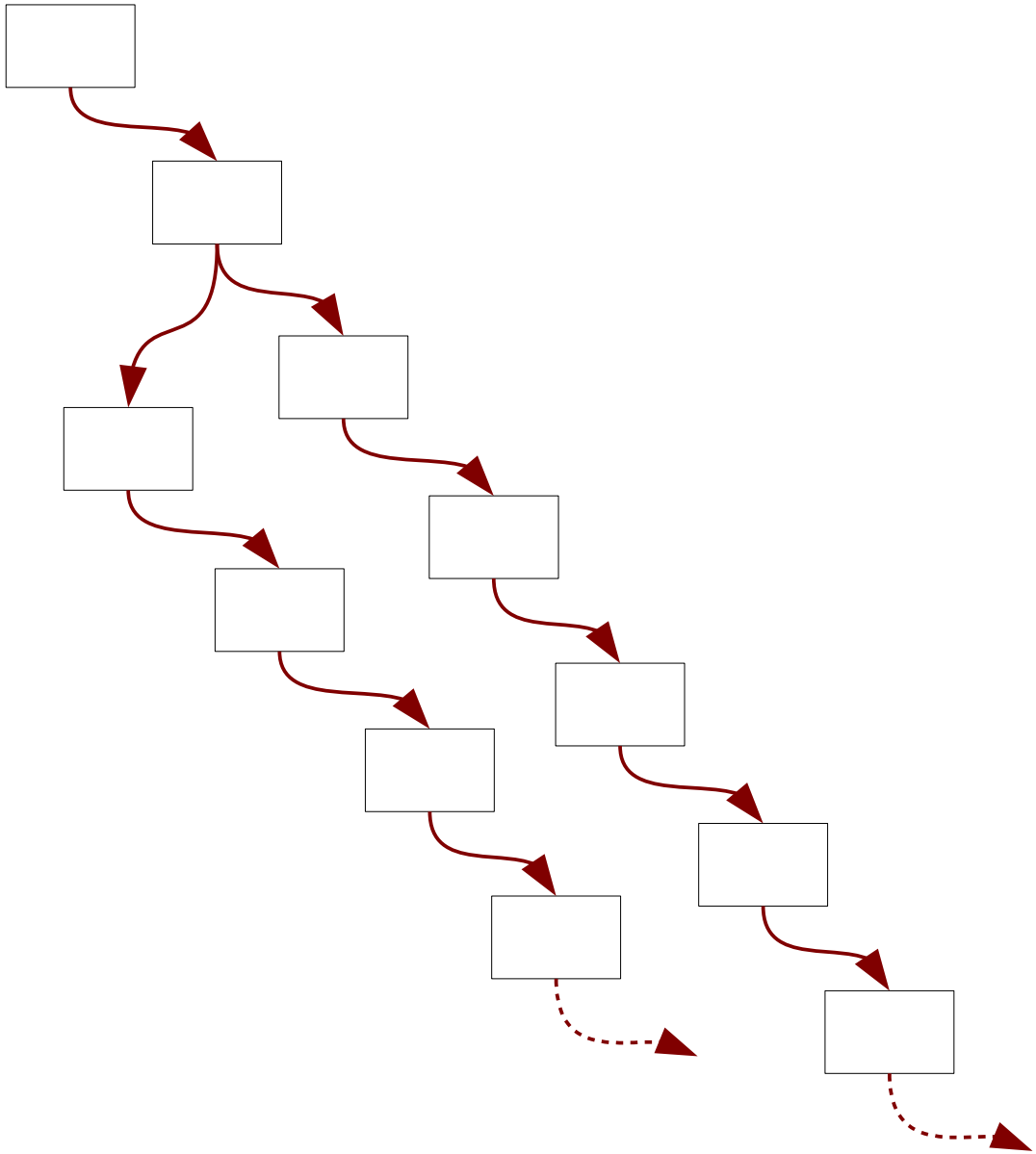
Alternative “pass the buck” class:

```
template <class Stream1, class Stream2>
class proxy
{
    Stream1 up_;
    Stream2 down_;
    vector<unsigned char> buffer1_, buffer2_;


public:
    proxy(...);
    Stream1& up() { return up_; }
    Stream2& down() { return down_; }

    template <class Handler>
    void async_run(Handler handler);
};
```

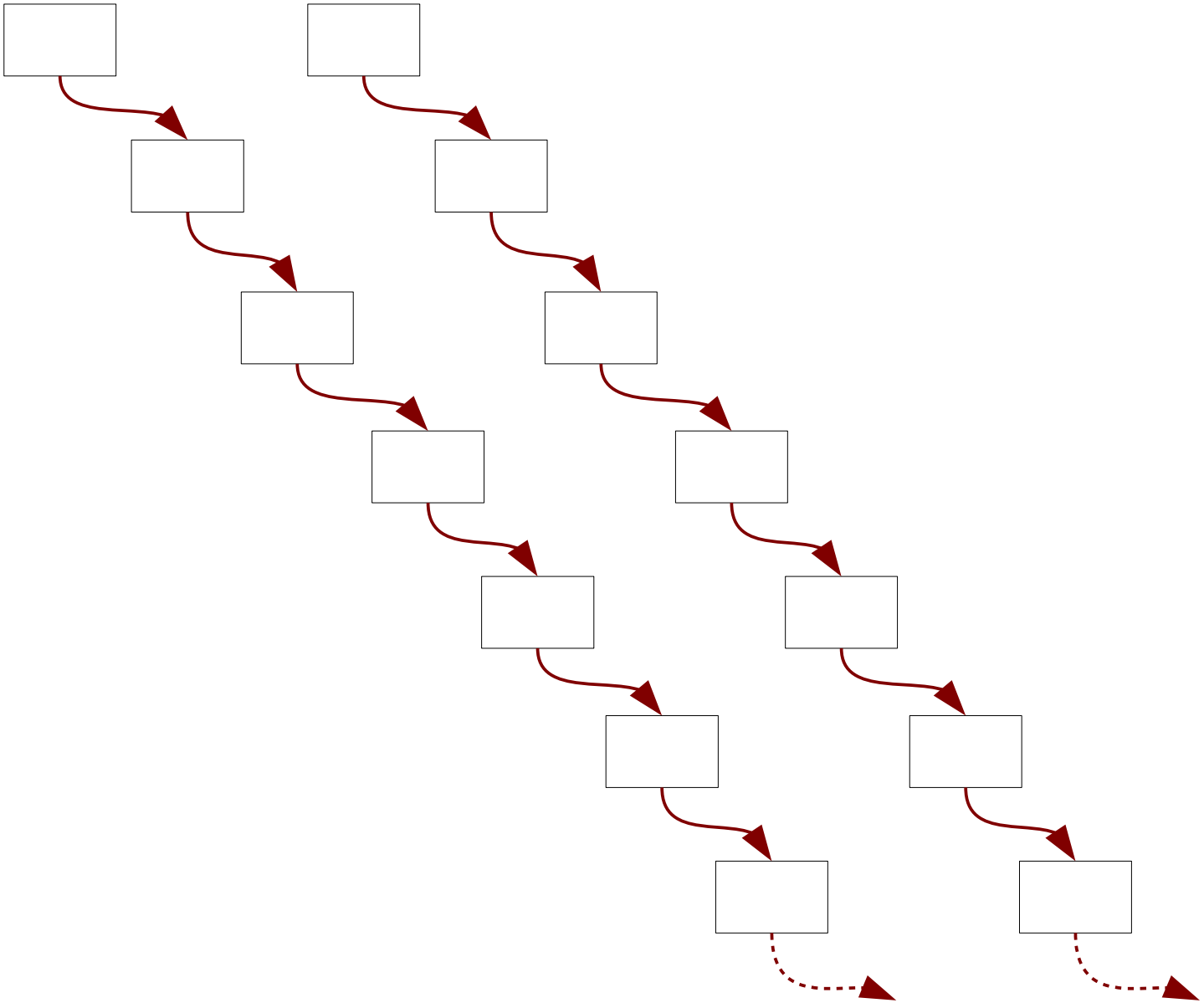
Managing Complexity



Managing Complexity



Managing Complexity



The buck stops here:

```
template <class Stream1, class Stream2>
class proxy : enable_shared_from_this<proxy<Stream1, Stream2> >
{
    asio::io_service strand_;
    Stream1 up_;
    Stream2 down_;
    vector<unsigned char> buffer1_, buffer2_;

    void do_start();
    void handle_transfer(const error_code& ec);

public:
    proxy(...);
    Stream1& up() { return up_; }
    Stream2& down() { return down_; }

    void start();
};
```

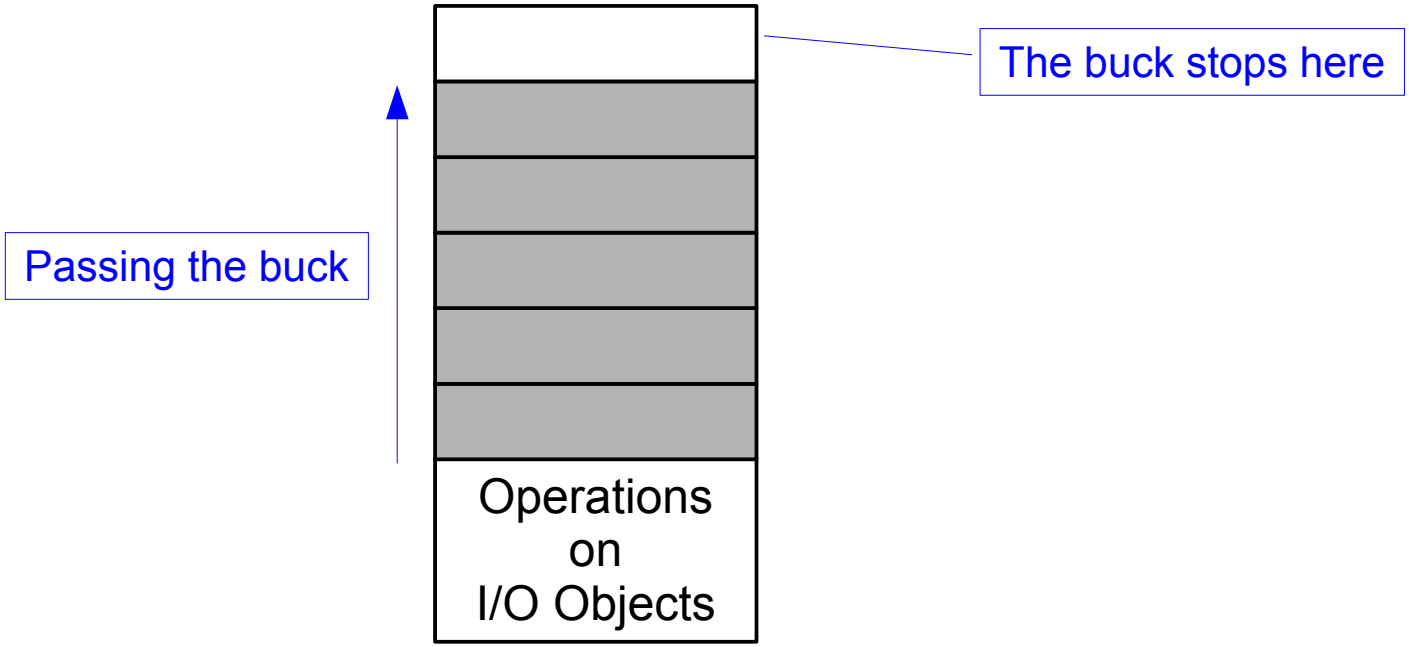
Managing Complexity

```
template <class Stream1, class Stream2>
void proxy::start()
{
    strand_.dispatch(
        bind(&proxy::do_start,
            this->shared_from_this()));
}

template <class Stream1, class Stream2>
void proxy::do_start()
{
    async_transfer(stream1_, stream2_, buffer1_,
        strand_.wrap(
            bind(&proxy::handle_transfer,
                this->shared_from_this(), _1)));
    async_transfer(stream2_, stream1_, buffer2_,
        strand_.wrap(
            bind(&proxy::handle_transfer,
                this->shared_from_this(), _1)));
}
```

Remind you of anything?

```
class connection :
    enable_shared_from_this<connection>
{
    // ...
    void start();
    void stop();
    bool is_stopped() const;
    // ...
};
```

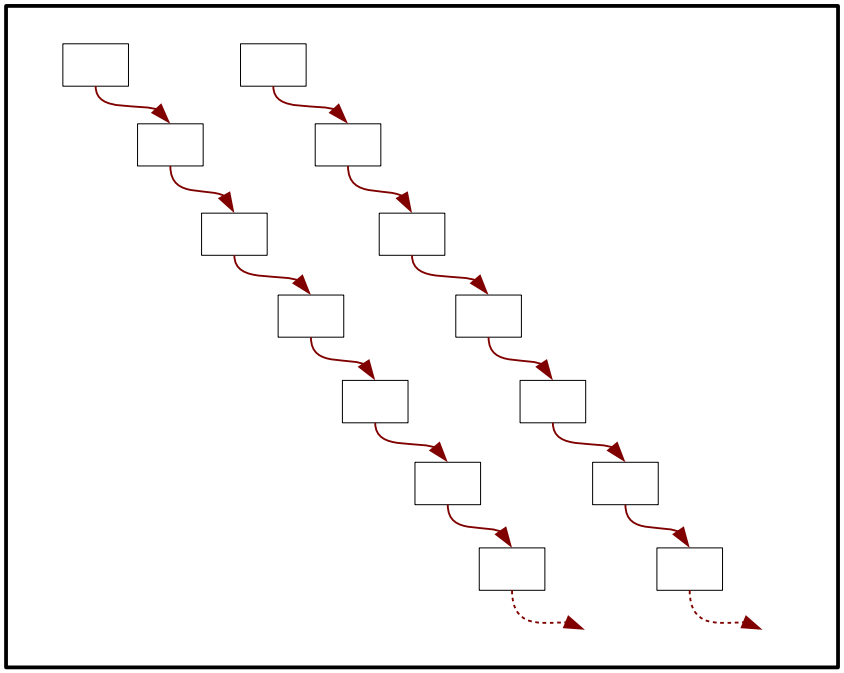
Managing Complexity



Managing Complexity

```
int main
{
    asio::io_service io_service;

    connection conn(io_service);

    io_service.run();
}
```

+



=

No reference counting
All memory committed up front
Possibility of zero allocations in steady state

Summary

Challenges:

- Object lifetimes
- Thinking asynchronously
- Threads
- Managing complexity

Guidelines:

- Know your object lifetime rules
- Assume asynchronous change, but know what's under your control
- Prefer to keep your logic single-threaded
- Pass the buck as often as you can