# Networking TS Workshop

## C++Now 2017

ciere consulting

Michael Caisse

mcaisse@ciere.com | follow @MichaelCaisse



ciere.com

# Part I

# Introduction

# Outline





ciere.com

# Be like Rust

**ubsan** 🙀 5:58 PM

I would like C++ to be at least as good as Rust is

# The Rust Evangelist

**ubsan** 10:24 AM

@sdowney that's why I like Rust's solution; 32-bit chars, but strings are utf-8

it works surprisingly well

although, of course, there's a conversion cost

(I hate saying stuff like that, because it looks like I'm advertising rust, but actually I just like its solution to a problem :/) (edited)

# Just Use Rust!

Three easy steps.

1. Make a cappuccino.
2. Write 2-lines of code
3. Mock people on C++ Slack channels (too cool for IRC)

# Just Use C++

Three easy steps.

1. Make a cappuccino.
2. Write a few lines of code.
3. Smile at the nice Rust people.

# Just Use C++

```cpp
#include <experimental/net>
#include <string>
#include <iostream>

namespace net = std::experimental::net;

int main()
{
   net::ip::tcp::iostream s("www.boost.org", "http");

   s << "GET / HTTP/1.0\r\n";
   s << "Host: www.boost.org\r\n";
   s << "Accept: */*\r\n";
   s << "Connection: close\r\n\r\n";

   std::string header;
   while(std::getline(s, header) && header != "\r")
      std::cout << header << "\n";

   std::cout << s.rdbuf();
}
```

# Stuff of old ...

- Atari MegaST
- NeXT Step
- IBM RS6000 - PowerPC Edition

# Boost.Asio 2010

# Slushies!

# I picked a winner!

Boost.Asio $\rightarrow$ Networking TS

# Networking TS

- Based on Boost.Asio by Christopher Kohlhoff
- Entered Boost March 2008
- March 2017 the 5th Draft of the Network TS published
- Issaquah 2016 - Unified Executors
- Standalone Asio implements much of the TS
- Chris has repository that snapshots Asio with proper names

# What is included

- Networking using TCP and UDP (including multicast)
- Client and server applications
- Scalability to handle many concurrent connections
- Protocol independence between IPv4 and IPv6
- Name resolution (i.e. DNS)
- Buffer management
- Timers

# What is not included

- Protocol implementations such as HTTP, SMTP or FTP
- Encryption (e.g. SSL, TLS)
- Operating system specific demultiplexing APIs
- Support for realtime environments
- QoS-enabled sockets
- Other TCP/IP protocols such as ICMP
- Functions and classes for enumerating network interfaces
- Other forms of asynchronous I/O, such as files

# Part II

# Stream

# Basic Stream Use

```cpp
#include <experimental/net>
#include <string>
#include <iostream>

namespace net = std::experimental::net;

int main()
{
   net::ip::tcp::iostream s("www.boost.org", "http");

   s << "GET / HTTP/1.0\r\n";
   s << "Host: www.boost.org\r\n";
   s << "Accept: */*\r\n";
   s << "Connection: close\r\n\r\n";

   std::string header;
   while(std::getline(s, header) && header != "\r")
      std::cout << header << "\n";

   std::cout << s.rdbuf();
}
```

# What Problems?

```
#include <experimental/net>
#include <string>
#include <iostream>

namespace net = std::experimental::net;

int main()
{
   net::ip::tcp::iostream s("www.boost.org", "http");

   s << "GET / HTTP/1.0\r\n";
   s << "Host: www.boost.org\r\n";
   s << "Accept: */*\r\n";
   s << "Connection: close\r\n\r\n";

   std::string header;
   while(std::getline(s, header) && header != "\r")
      std::cout << header << "\n";

   std::cout << s.rdbuf();
}
```

# Add error checking

```
net::ip::tcp::iostream s;


s.expires_after(5s);
s.connect("www.boost.org", "https");


if(!s)
{
   std::cout << "error: " << s.error().message() << std::endl;
   return -1;
}


s << "GET / HTTP/1.0\r\n";
s << "Host: www.boost.org\r\n";
s << "Accept: */*\r\n";
s << "Connection: close\r\n\r\n";

std::string header;
while(s && std::getline(s, header) && header != "\r")
   std::cout << header << "\n";

std::cout << s.rdbuf();
```

# Add error checking

```
net::ip::tcp::iostream s;


s.expires_after(5s);
s.connect("www.boost.org", "https");


if(!s)
{
    std::cout << "error: " << s.error().message() << std::endl;
    return -1;
}


s << "GET / HTTP/1.0\r\n";
s << "Host: www.boost.org\r\n";
s << "Accept: */*\r\n";
s << "Connection: close\r\n\r\n";

std::string header;
while(s && std::getline(s, header) && header != "\r")
    std::cout << header << "\n";

std::cout << s.rdbuf();
```

# std::error_code

```
if(!s)
{
    std::cout << "error: " << s.error().message() << std::endl;
    return -1;
}
```

- `std::error_code` entered with C++11
- Based on Boost library
- Captures platform specific error
- Holds a pointer to the `error_category`, generic mapping

# T.T.H.W.

Streams satisfy the T.T.H.W (Time To Hello World) measurement.

```cpp
#include <experimental/net>
#include <string>
#include <iostream>

namespace net = std::experimental::net;

int main()
{
   net::ip::tcp::iostream s("www.boost.org", "http");

   s << "GET / HTTP/1.0\r\n";
   s << "Host: www.boost.org\r\n";
   s << "Accept: */*\r\n";
   s << "Connection: close\r\n\r\n";

   std::string header;
   while(std::getline(s, header) && header != "\r")
      std::cout << header << "\n";

   std::cout << s.rdbuf();
}
```

# Part III

# Synchronous

# Asynchronous versus Synchronous

Daughter #1

me: “Please make me a coffee.”
daughter: “Sure Dad”

*time passes ... I work. She makes a cappuccino.*

daughter: “Here is your coffee.”
me: “Thanks”

# Asynchronous versus Synchronous

Daughter #3

me: “Please make me a coffee.”
daughter: “I would love to!”

*we both walk to the machine. I supervise (watch). She makes a cappuccino.*

daughter: “Here is your coffee.”
me: “Thanks”

# Synchronous

When should synchronous I/O be used?

# Synchronous

- Often simple programs
- Default (system level) timeouts meet need
- Fine grained control, with keen understanding of operations blocking

# Synchronous Example

```
net::io_context io_context;
tcp::socket socket(io_context);
tcp::resolver resolver(io_context);

net::connect(socket,
             resolver.resolve("www.boost.org", "http"));

for(auto v : { "GET / HTTP/1.0\r\n"s
             , "Host: www.boost.org\r\n"s
             , "Accept: */*\r\n"s
             , "Connection: close\r\n\r\n"s } )
{
   net::write(socket, net::buffer(v));
}

std::string header;
net::read_until(socket,
                net::dynamic_buffer(header),
                "\r\n\r\n");

std::error_code e;
std::string body;

net::read(socket,
          net::dynamic_buffer(body),
          e);

std::cout << "Header:\n" << header << "Body:\n" << body
          << "Error code: " << e.message() << std::endl;
```

# Synchronous Example

```
net::io_context io_context;
tcp::socket socket(io_context);
tcp::resolver resolver(io_context);

net::connect(socket,
             resolver.resolve("www.boost.org", "http"));

for(auto v : { "GET / HTTP/1.0\r\n"s
             , "Host: www.boost.org\r\n"s
             , "Accept: */*\r\n"s
             , "Connection: close\r\n\r\n"s } )
{
   net::write(socket, net::buffer(v));
}

std::string header;
net::read_until(socket,
                net::dynamic_buffer(header),
                "\r\n\r\n");

std::error_code e;
std::string body;

net::read(socket,
          net::dynamic_buffer(body),
          e);

std::cout << "Header:\n" << header << "Body:\n" << body
          << "Error code: " << e.message() << std::endl;
```

# Synchronous Example

```
net::io_context io_context;
tcp::socket socket(io_context);
tcp::resolver resolver(io_context);

net::connect(socket,
             resolver.resolve("www.boost.org", "http"));

for(auto v : { "GET / HTTP/1.0\r\n"s
             , "Host: www.boost.org\r\n"s
             , "Accept: */*\r\n"s
             , "Connection: close\r\n\r\n"s } )
{
   net::write(socket, net::buffer(v));
}

std::string header;
net::read_until(socket,
                net::dynamic_buffer(header),
                "\r\n\r\n");

std::error_code e;
std::string body;

net::read(socket,
          net::dynamic_buffer(body),
          e);

std::cout << "Header:\n" << header << "Body:\n" << body
          << "Error code: " << e.message() << std::endl;
```

# Synchronous Example

```
net::io_context io_context;
tcp::socket socket(io_context);
tcp::resolver resolver(io_context);

net::connect(socket,
             resolver.resolve("www.boost.org", "http"));

for(auto v : { "GET / HTTP/1.0\r\n"s
             , "Host: www.boost.org\r\n"s
             , "Accept: */*\r\n"s
             , "Connection: close\r\n\r\n"s } )
{
   net::write(socket, net::buffer(v));
}

std::string header;
net::read_until(socket,
                net::dynamic_buffer(header),
                "\r\n\r\n");

std::error_code e;
std::string body;

net::read(socket,
          net::dynamic_buffer(body),
          e);

std::cout << "Header:\n" << header << "Body:\n" << body
          << "Error code: " << e.message() << std::endl;
```

# Synchronous Example

```
net::io_context io_context;
tcp::socket socket(io_context);
tcp::resolver resolver(io_context);

net::connect(socket,
             resolver.resolve("www.boost.org", "http"));

for(auto v : { "GET / HTTP/1.0\r\n"s
             , "Host: www.boost.org\r\n"s
             , "Accept: */*\r\n"s
             , "Connection: close\r\n\r\n"s } )
{
   net::write(socket, net::buffer(v));
}

std::string header;
net::read_until(socket,
                net::dynamic_buffer(header),
                "\r\n\r\n");

std::error_code e;
std::string body;

net::read(socket,
          net::dynamic_buffer(body),
          e);

std::cout << "Header:\n" << header << "Body:\n" << body
          << "Error code: " << e.message() << std::endl;
```

# Synchronous Example

```
net::io_context io_context;
tcp::socket socket(io_context);
tcp::resolver resolver(io_context);

net::connect(socket,
             resolver.resolve("www.boost.org", "http"));

for(auto v : { "GET / HTTP/1.0\r\n"s
             , "Host: www.boost.org\r\n"s
             , "Accept: */*\r\n"s
             , "Connection: close\r\n\r\n"s } )
{
   net::write(socket, net::buffer(v));
}

std::string header;
net::read_until(socket,
                net::dynamic_buffer(header),
                "\r\n\r\n");

std::error_code e;
std::string body;

net::read(socket,
          net::dynamic_buffer(body),
          e);

std::cout << "Header:\n" << header << "Body:\n" << body
          << "Error code: " << e.message() << std::endl;
```

# Synchronous Example

```
net::io_context io_context;
tcp::socket socket(io_context);
tcp::resolver resolver(io_context);

net::connect(socket,
             resolver.resolve("www.boost.org", "http"));

for(auto v : { "GET / HTTP/1.0\r\n"s
             , "Host: www.boost.org\r\n"s
             , "Accept: */*\r\n"s
             , "Connection: close\r\n\r\n"s } )
{
   net::write(socket, net::buffer(v));
}

std::string header;
net::read_until(socket,
                net::dynamic_buffer(header),
                "\r\n\r\n");

std::error_code e;
std::string body;

net::read(socket,
          net::dynamic_buffer(body),
          e);

std::cout << "Header:\n" << header << "Body:\n" << body
          << "Error code: " << e.message() << std::endl;
```

# net::io_context

```
net::io_context io_context;
tcp::socket socket(io_context);
tcp::resolver resolver(io_context);
```

`net::io_context` is similar to `asio::io_service`

The `io_context` is provided to each of the networking objects.

*more on this later...*

# tcp::socket

```
net::io_context io_context;
tcp::socket socket(io_context);
tcp::resolver resolver(io_context);
```

`tcp` meets the requirement for an `InternetProtocol`

There is also `udp`

# tcp::socket

```
net::io_context io_context;
tcp::socket socket(io_context);
tcp::resolver resolver(io_context);
```

`tcp::socket` is a `basic_stream_socket<tcp>`

`udp::socket` is a `basic_datagram_socket<udp>`

# tcp::socket

```
net::io_context io_context;
tcp::socket socket(io_context);
tcp::resolver resolver(io_context);
```

`tcp::socket` is a `basic_stream_socket<tcp>`

`udp::socket` is a `basic_datagram_socket<udp>`

# tcp::resolver

```
net::io_context io_context;
tcp::socket socket(io_context);
tcp::resolver resolver(io_context);
```

`tcp::resolver` is a `basic_resolver<tcp>`

# tcp::resolver

```
net::io_context io_context;
tcp::socket socket(io_context);
tcp::resolver resolver(io_context);
```

A `resolver` performs name resolution.

host name and service name → `end_point`

-or-

`end_point` → host name and service name

# net::connect

```
net::connect(socket,
             resolver.resolve("www.boost.org", "http"));
```

Where is the error handling?

# net::connect

```
net::connect(socket,
             resolver.resolve("www.boost.org", "http"));
```

Where is the error handling?

# Error Handling

Synchronous functions have signatures like:

```
R f (A1 a1, A2 a2, ..., AN aN);
R f (A1 a1, A2 a2, ..., AN aN, error_code& ec);
```

# Error Handling

```
R f (A1 a1, A2 a2, ..., AN aN);
```

Throws an exception matching the type of `system_error`

# Error Handling

```
R f (A1 a1, A2 a2, ..., AN aN, error_code& ec);
```

Sets the `error_code` appropriately on error; otherwise `!ec` is `true`

# net::connect

```
net::connect(socket,
             resolver.resolve("www.boost.org", "http"));
```

# net::write

```
for(auto v : { "GET / HTTP/1.0\r\n"s
             , "Host: www.boost.org\r\n"s
             , "Accept: */*\r\n"s
             , "Connection: close\r\n\r\n"s } )
{
   net::write(socket, net::buffer(v));
}
```

# Buffers

- All I/O is performed through buffers
- Buffers **do not** own the actual memory

# Buffers

Three types of buffers:

1. Mutable Buffer Sequence
2. Constant Buffer Sequence
3. Dynamic Buffer

Use a creation function to get a buffer.

# Mutable Buffer Sequence Creation

```
mutable_buffer buffer(void* p, size_t n) noexcept;
mutable_buffer buffer(const mutable_buffer& b) noexcept;
mutable_buffer buffer(const mutable_buffer& b, size_t n) noexcept;

template<class T, size_t N>
mutable_buffer buffer(T (&data)[N]) noexcept;

template<class T, size_t N>
mutable_buffer buffer(array<T, N>& data) noexcept;

template<class T, class Allocator>
mutable_buffer buffer(vector<T, Allocator>& data) noexcept;

template<class CharT, class Traits, class Allocator>
mutable_buffer buffer(basic_string<CharT, Traits, Allocator>& data) noexcept;

template<class T, size_t N>
mutable_buffer buffer(T (&data)[N], size_t n) noexcept;

template<class T, size_t N>
mutable_buffer buffer(array<T, N>& data, size_t n) noexcept;

template<class T, class Allocator>
mutable_buffer buffer(vector<T, Allocator>& data, size_t n) noexcept;

template<class CharT, class Traits, class Allocator>
mutable_buffer buffer(basic_string<CharT, Traits, Allocator>& data, size_t n) no
```

# Constant Buffer Sequence Creation

```
const_buffer buffer(const void* p, size_t n) noexcept;
const_buffer buffer(const const_buffer& b) noexcept;
const_buffer buffer(const const_buffer& b, size_t n) noexcept;

template<class T, size_t N>
const_buffer buffer(const T (&data)[N]) noexcept;

template<class T, size_t N>
const_buffer buffer(array<const T, N>& data) noexcept;

template<class T, size_t N>
const_buffer buffer(const array<T, N>& data) noexcept;

template<class T, class Allocator>
const_buffer buffer(const vector<T, Allocator>& data) noexcept;

template<class CharT, class Traits, class Allocator>
const_buffer buffer(const basic_string<CharT, Traits, Allocator>& data) noexcept

template<class CharT, class Traits>
const_buffer buffer(basic_string_view<CharT, Traits> data) noexcept;

template<class T, size_t N>
const_buffer buffer(const T (&data)[N], size_t n) noexcept;

template<class T, size_t N>
const_buffer buffer(array<const T, N>& data, size_t n) noexcept;

template<class T, size_t N>
const_buffer buffer(const array<T, N>& data, size_t n) noexcept;
```

# Dynamic Buffer Creation

Methods that take a `size_t` are setting the max size.

```
template<class T, class Allocator>
dynamic_vector_buffer<T, Allocator>
dynamic_buffer(vector<T, Allocator>& vec) noexcept;

template<class T, class Allocator>
dynamic_vector_buffer<T, Allocator>
dynamic_buffer(vector<T, Allocator>& vec, size_t n) noexcept;

template<class CharT, class Traits, class Allocator>
dynamic_string_buffer<CharT, Traits, Allocator>
dynamic_buffer(basic_string<CharT, Traits, Allocator>& str) noexcept;

template<class CharT, class Traits, class Allocator>
dynamic_string_buffer<CharT, Traits, Allocator>
dynamic_buffer(basic_string<CharT, Traits, Allocator>& str, size_t n) noexce
```

# Dynamic Buffer Creation

Methods that take a `size_t` are setting the max size.

```
template<class T, class Allocator>
dynamic_vector_buffer<T, Allocator>
dynamic_buffer(vector<T, Allocator>& vec) noexcept;

template<class T, class Allocator>
dynamic_vector_buffer<T, Allocator>
dynamic_buffer(vector<T, Allocator>& vec, size_t n) noexcept;

template<class CharT, class Traits, class Allocator>
dynamic_string_buffer<CharT, Traits, Allocator>
dynamic_buffer(basic_string<CharT, Traits, Allocator>& str) noexcept;

template<class CharT, class Traits, class Allocator>
dynamic_string_buffer<CharT, Traits, Allocator>
dynamic_buffer(basic_string<CharT, Traits, Allocator>& str, size_t n) noexce
```

# net::write

```
for(auto v : { "GET / HTTP/1.0\r\n"s
             , "Host: www.boost.org\r\n"s
             , "Accept: */*\r\n"s
             , "Connection: close\r\n\r\n"s } )
{
   net::write(socket, net::buffer(v));
}
```

# net::read_until

## Synchronous delimited read

```
std::string header;
net::read_until(socket,
                net::dynamic_buffer(header),
                "\r\n\r\n");
```

- reads until delimiter is contained in buffer
- delimiter is a `char` or a `string_view`

# net::read_until

Synchronous delimited read

```
std::string header;
net::read_until(socket,
                net::dynamic_buffer(header),
                "\r\n\r\n");
```

- reads until delimiter is contained in buffer
- delimiter is a `char` or a `string_view`

# net::read_until

Synchronous delimited read

```
std::string header;
net::read_until(socket,
                net::dynamic_buffer(header),
                "\r\n\r\n");
```

- reads until delimiter is contained in buffer
- delimiter is a `char` or a `string_view`

# net::read_until

```
std::string header;
net::read_until(socket,
                net::dynamic_buffer(header),
                "\r\n\r\n");
```

- How are errors being handled?
- How much memory will be consumed?

# net::read_until

```
std::string header;
net::read_until(socket,
                net::dynamic_buffer(header),
                "\r\n\r\n");
```

- How are errors being handled?
- How much memory will be consumed?

# net::read

```
std::error_code e;
std::string body;

net::read(socket,
          net::dynamic_buffer(body),
          e);
```

What is this one doing?

# net::read

```
std::error_code e;
std::string body;

net::read(socket,
          net::dynamic_buffer(body),
          e);
```

What is this one doing?

# Demo

Run it!

# net::read_until revisited

```
std::string header;
net::read_until(socket,
                net::dynamic_buffer(header),
                "\r\n\r\n");
```

- reads until delimiter is contained in buffer

Reads via `read_some`

# net::read with CompletionCondition

Replace:

```
std::string header;
net::read_until(socket,
                net::dynamic_buffer(header),
                "\r\n\r\n");
```

with:

```
net::read(socket,
          net::dynamic_buffer(header),
          CompletionCondition);
```

# net::read with CompletionCondition

```
net::read(socket,
          net::dynamic_buffer(header),
          CompletionCondition);
```

- CompletionCondition is called prior to each `read_some`
- Signature of:

```
std::size_t func(error_code, std::size_t total_read);
```

- returns the number of bytes to be read on the next `read_some` call

# `net::read` with CompletionCondition

```
net::read(socket,
          net::dynamic_buffer(header),
          CompletionCondition);
```

`net::read` continues to transfer bytes until one of:

- there is an error
- CompletionCondition returns 0
- `buffer_size(buffers)` is transferred

# net::read with CompletionCondition

Replace:

```
std::string header;
net::read_until(socket,
                net::dynamic_buffer(header),
                "\r\n\r\n");
```

with:

```
std::string header;
net::read(socket,
          net::dynamic_buffer(header),
          [&header](auto ec, auto n) -> std::size_t
          {
              if(ec ||
                  (   header.size() > 3
                   && header.compare(header.size()-4, 4,
                                     "\r\n\r\n") == 0 ))
              {
                  return 0;
              }
              return 1;
          });
```

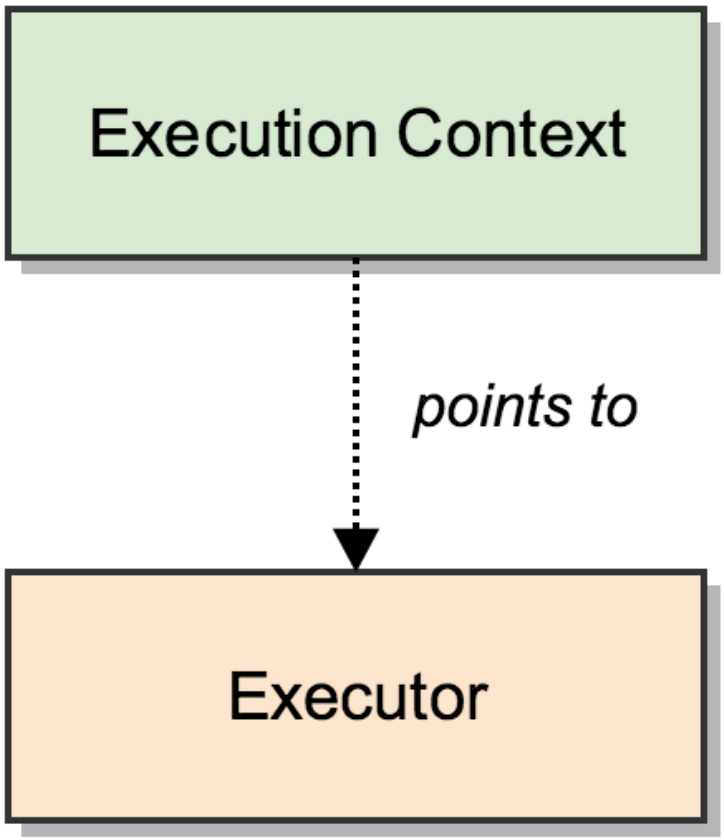
# Demo with CompletionCondition

Demo

# Part IV

# Asynchronous

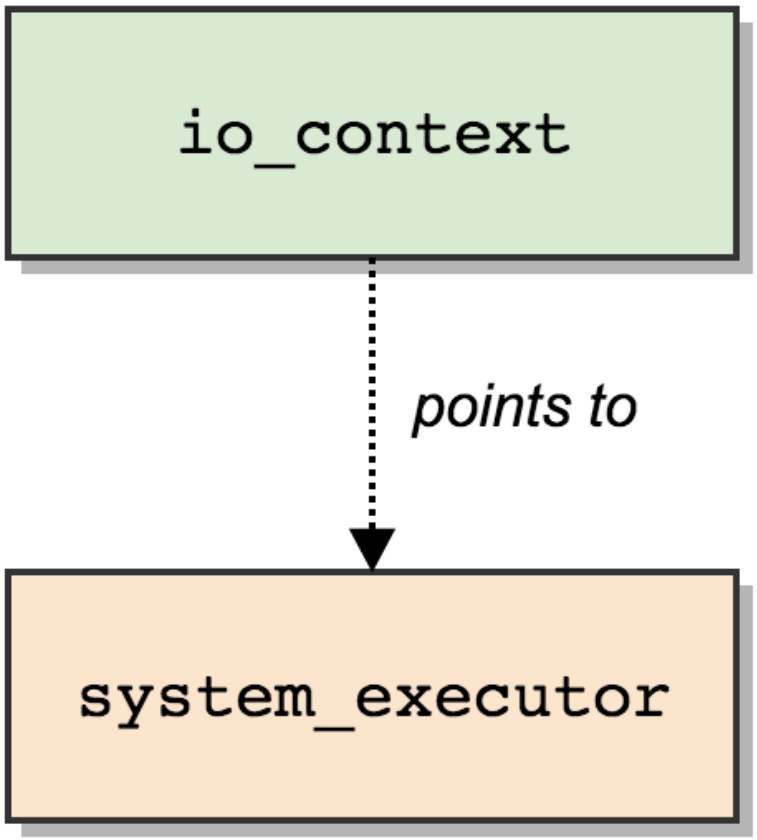
# Revisit `io_context`




`io_context`

Embodies rules for determining how function objects are executed.

# Revisit `io_context`



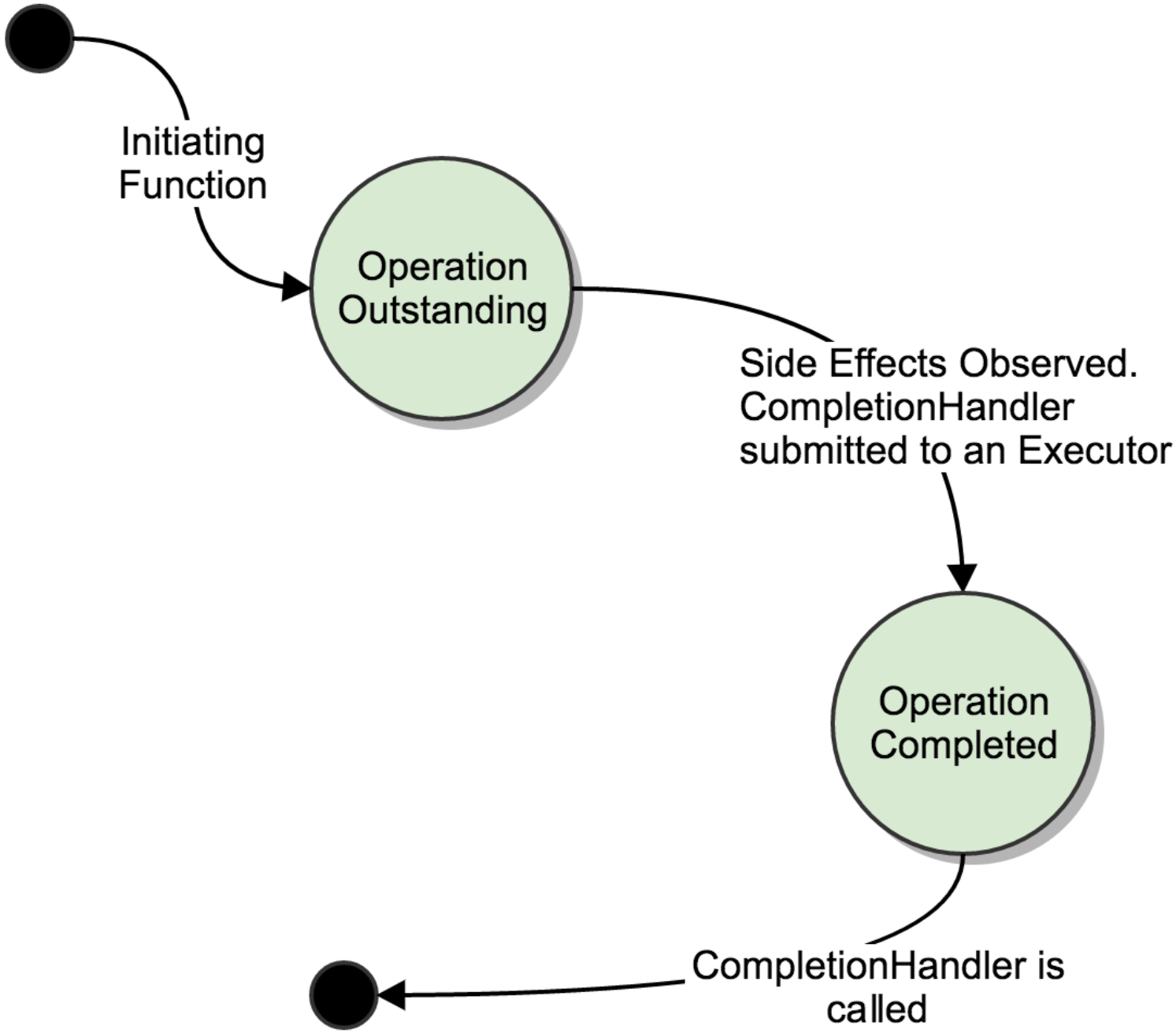

represents a set of rules where function objects are permitted to execute on any thread

ciere.com

# Live Coding ...

`execution_context` and posting work.

# Asynchronous Operation



# Asynchronous Operation

`async_*` functions are intiating functions.

More live coding ... gasp!

# Async Example - Future

```
net::io_context io_context;
std::thread t([&io_context](){io_context.run();});

auto resolver = tcp::resolver(io_context);
auto resolve = resolver.async_resolve("www.boost.org", "http",
                                      net::use_future);

tcp::socket socket(io_context);
auto connect = net::async_connect(socket, resolve.get(),
                                  net::use_future);

auto request = "GET / HTTP/1.0\r\nHost: www.boost.org\r\n"
               "Accept: */*\r\nConnection: close\r\n\r\n"s;

socket.async_send(net::buffer(request), [](auto, auto){});

std::string header;
auto header_read = net::async_read_until(socket, net::dynamic_buffer(header),
                                         "\r\n\r\n",
                                         net::user_future);
// do some stuff ....
if(header_read.get() <= 2) { std::cout << "no header\n"; }

std::string body;
auto body_read = net::async_read(socket, net::dynamic_buffer(body),
                                 net::transfer_all(),
                                 net::use_future);
// do some stuff ....
body_read.get();
std::cout << "Header:\n" << header << "Body:\n" << body << "\n";
t.join();
```

# Async Example - main

```
int main()
{
    net::io_context io_context;
    auto work = net::make_work_guard(io_context);
    std::thread t([&io_context](){io_context.run();});

    web_page_getter wpg(io_context);

    auto f = wpg.get_page("www.boost.org", "http");

    try
    {
        auto result = f.get();
        std::cout << "Header:\n" << std::get<0>(result)
                  << "Body:\n" << std::get<1>(result) << std::endl;
    }
    catch(std::error_code const & ec)
    {
        std::cout << "Error: " << ec.message() << std::endl;
    }

    io_context.stop();
    t.join();
}
```

# Async Example - main

```
int main()
{
    net::io_context io_context;
    auto work = net::make_work_guard(io_context);
    std::thread t([&io_context](){io_context.run();});

    web_page_getter wpg(io_context);

    auto f = wpg.get_page("www.boost.org", "http");

    try
    {
        auto result = f.get();
        std::cout << "Header:\n" << std::get<0>(result)
                  << "Body:\n" << std::get<1>(result) << std::endl;
    }
    catch(std::error_code const & ec)
    {
        std::cout << "Error: " << ec.message() << std::endl;
    }

    io_context.stop();
    t.join();
}
```

# Async Example - main

```
int main()
{
    net::io_context io_context;
    auto work = net::make_work_guard(io_context);
    std::thread t([&io_context](){io_context.run();});

    web_page_getter wpg(io_context);

    auto f = wpg.get_page("www.boost.org", "http");

    try
    {
        auto result = f.get();
        std::cout << "Header:\n" << std::get<0>(result)
                  << "Body:\n" << std::get<1>(result) << std::endl;
    }
    catch(std::error_code const & ec)
    {
        std::cout << "Error: " << ec.message() << std::endl;
    }

    io_context.stop();
    t.join();
}
```

# Async Example - main

```
int main()
{
    net::io_context io_context;
    auto work = net::make_work_guard(io_context);
    std::thread t([&io_context](){io_context.run();});

    web_page_getter wpg(io_context);

    auto f = wpg.get_page("www.boost.org", "http");

    try
    {
        auto result = f.get();
        std::cout << "Header:\n" << std::get<0>(result)
                  << "Body:\n" << std::get<1>(result) << std::endl;
    }
    catch(std::error_code const & ec)
    {
        std::cout << "Error: " << ec.message() << std::endl;
    }

    io_context.stop();
    t.join();
}
```

# Async Example - main

```
int main()
{
    net::io_context io_context;
    auto work = net::make_work_guard(io_context);
    std::thread t([&io_context](){io_context.run();});

    web_page_getter wpg(io_context);

    auto f = wpg.get_page("www.boost.org", "http");

    try
    {
        auto result = f.get();
        std::cout << "Header:\n" << std::get<0>(result)
                  << "Body:\n" << std::get<1>(result) << std::endl;
    }
    catch(std::error_code const & ec)
    {
        std::cout << "Error: " << ec.message() << std::endl;
    }

    io_context.stop();
    t.join();
}
```

# Async Example - main

```
int main()
{
   net::io_context io_context;
   auto work = net::make_work_guard(io_context);
   std::thread t([&io_context](){io_context.run();});

   web_page_getter wpg(io_context);

   auto f = wpg.get_page("www.boost.org", "http");

   try
   {
      auto result = f.get();
      std::cout << "Header:\n" << std::get<0>(result)
                << "Body:\n" << std::get<1>(result) << std::endl;
   }
   catch(std::error_code const & ec)
   {
      std::cout << "Error: " << ec.message() << std::endl;
   }

   io_context.stop();
   t.join();
}
```

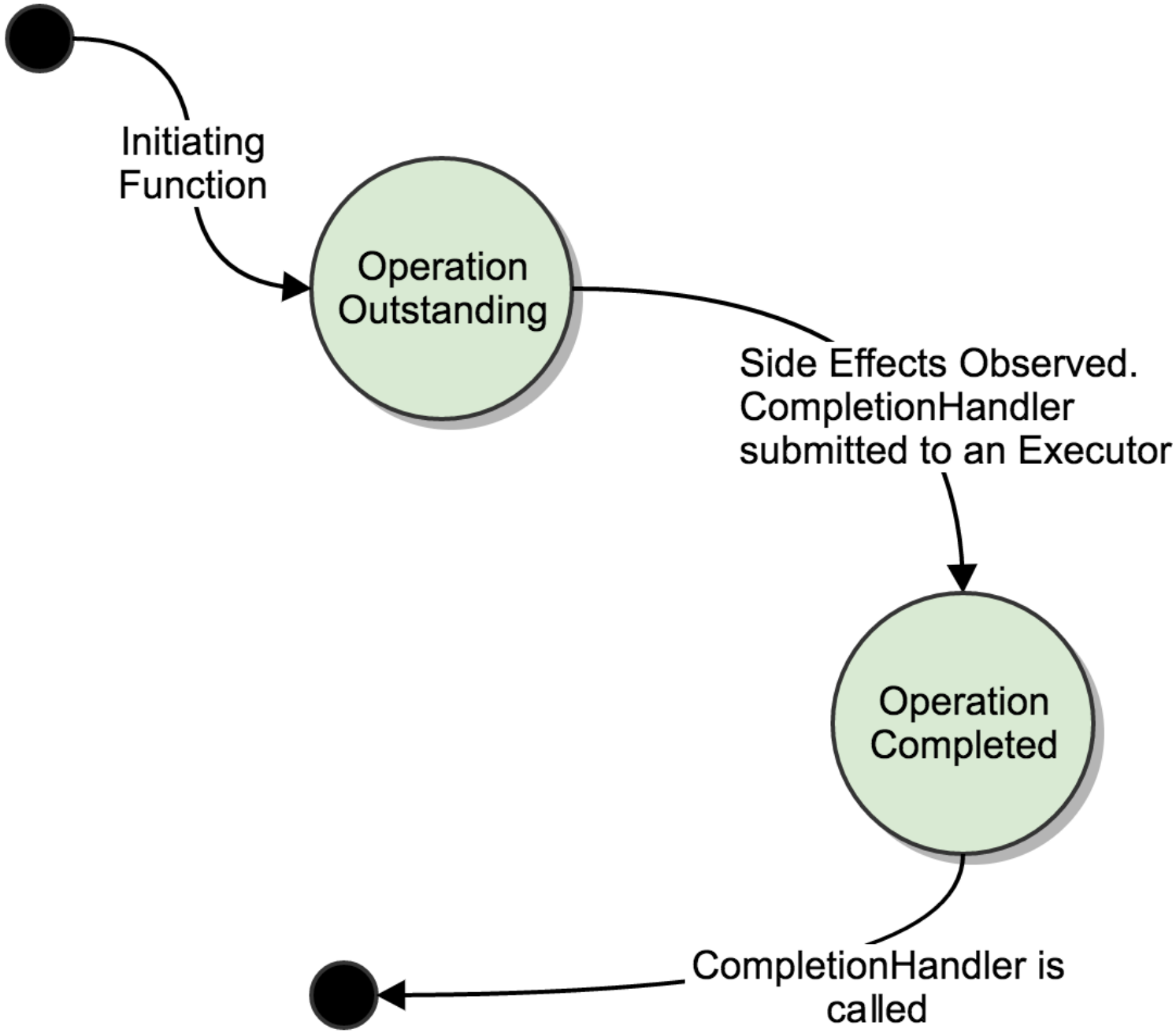
# Async Example - web_page_getter

```
class web_page_getter
{
public:
   using page_type = std::tuple<std::string, std::string>;

   web_page_getter(net::io_context & io_context);

   std::future<page_type> get_page(std::string const & host,
                                   std::string const & port);

private:
   void connect(std::string const & host, std::string const & port);
   void read_header();
   void read_body();
   void send_request();

private:
   net::io_context & io_context_;
   tcp::socket socket_;
   std::string header_;
   std::string body_;
   std::promise<page_type> promise_;
};
```

# Async Example - web_page_getter

```
class web_page_getter
{
public:
   using page_type = std::tuple<std::string, std::string>;

   web_page_getter(net::io_context & io_context);

   std::future<page_type> get_page(std::string const & host,
                                   std::string const & port);

private:
   void connect(std::string const & host, std::string const & port);
   void read_header();
   void read_body();
   void send_request();

private:
   net::io_context & io_context_;
   tcp::socket socket_;
   std::string header_;
   std::string body_;
   std::promise<page_type> promise_;
};
```

# Chain in the CompletionHandler



# Async Example - web_page_getter

```
web_page_getter(net::io_context & io_context)
   : io_context_(io_context)
   , socket_(io_context_)
{}
```

# Async Example - web_page_getter

```
std::future<page_type> get_page(std::string const & host,
                                std::string const & port)
{
   promise_ = std::promise<page_type>{};
   socket_ = tcp::socket{io_context_};
   header_ = std::string{};
   body_ = std::string{};

   connect(host, port);

   return promise_.get_future();
}
```

# Async Example - web_page_getter

```
void connect(std::string const & host,
             std::string const & port)
{
   tcp::resolver resolver(io_context_);
   net::async_connect(socket_,
                      resolver.resolve(host, port),
                      [this](auto ec, auto end_point)
                      {
                         if(!ec)
                         {
                            send_request();
                            read_header();
                         }
                         else
                         {
                            promise_.set_exception(
                               std::make_exception_ptr(ec)
                               );
                         }
                      });
}
```

# Async Example - web_page_getter

```
void send_request()
{
    auto request = "GET / HTTP/1.0\r\n"
                   "Host: www.boost.org\r\n"
                   "Accept: */*\r\n"
                   "Connection: close\r\n\r\n"s;

    socket_.async_send(net::buffer(request),
                       [](auto, auto){} );
}
```

# Async Example - web_page_getter

```
void read_header()
{
    net::async_read_until(socket_,
                          net::dynamic_buffer(header_),
                          "\r\n\r\n",
                          [this](auto ec, auto bytes_trans)
                          {
                              if(!ec)
                              {
                                  read_body();
                              }
                              else
                              {
                                  promise_.set_exception(
                                      std::make_exception_ptr(ec)
                                      );
                              }
                          });
}
```

# Async Example - web_page_getter

```
void read_body()
{
    net::async_read(socket_,
                    net::dynamic_buffer(body_),
                    net::transfer_all(),
                    [this](auto ec, auto bytes_trans)
                    {
                        if(!ec || ec == net::stream_errc::eof )
                        {
                            promise_.set_value(
                                page_type{std::move(header_),
                                          std::move(body_)}
                                );
                        }
                        else
                        {
                            promise_.set_exception(
                                std::make_exception_ptr(ec)
                                );
                        }
                        std::error_code close_ec;
                        socket_.close(close_ec);
                    });
}
```

# Do Not Go Crazy!

26

## Is there efficient alternative to ASIO? (self.cpp)

submitted 3 months ago by Z01dbrg

I know ASIO is great according to some people, and I agree that it is better than hand written crap code, but I really really dislike that style of programming in a language without GC. For more examples of what I dislike you have this video filled with examples of Boost.ASIO. https://www.youtube.com/watch?v=rwOv_tw2eA4 This is not about GC being a generally positive thing, I just feel that in case of async it is much nicer than manual shared_ptring for the emulation of GC.

So to be clear I know that 1000 threads can handle 1000 connections, and I can do simple blocking code with those 1000 threads, I am just asking is there efficient alternative to ASIO...

For example could some VS co_await(abomination of a keyword) library for networking be as good as ASIO, and if so is there such library?

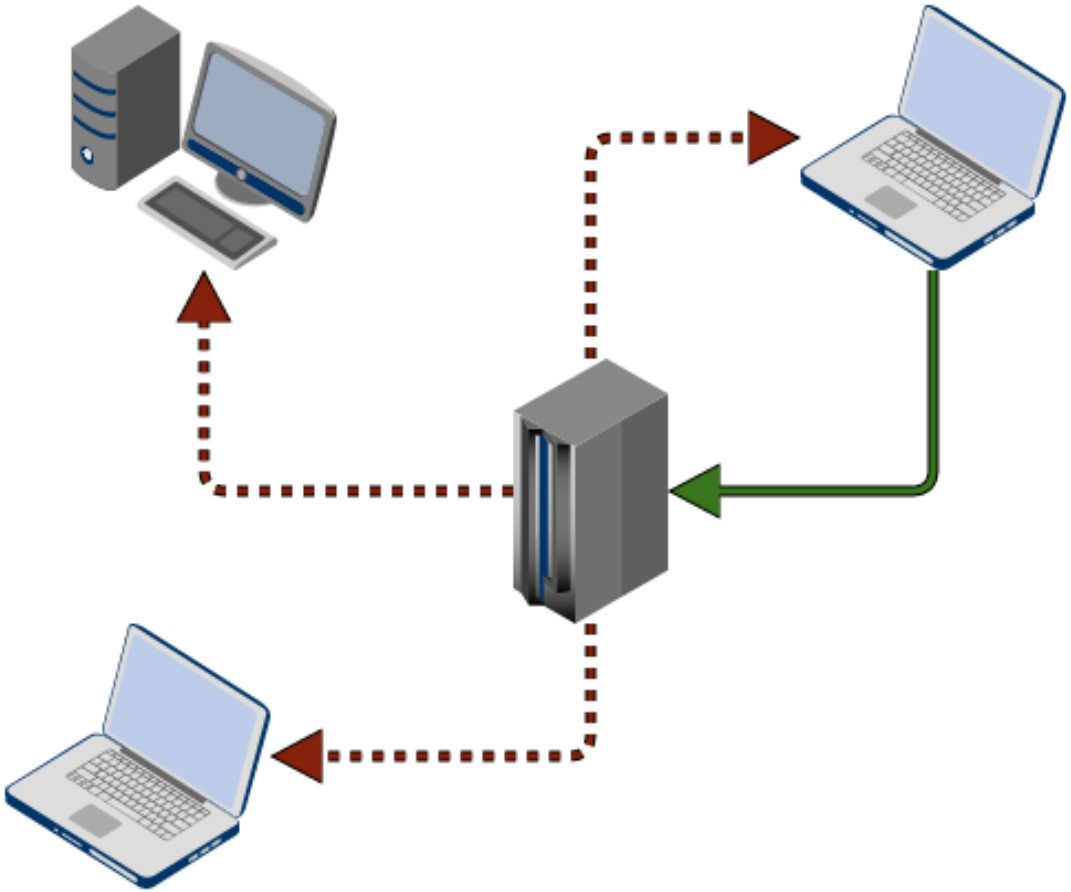
52 comments share save hide give gold report

# Part V

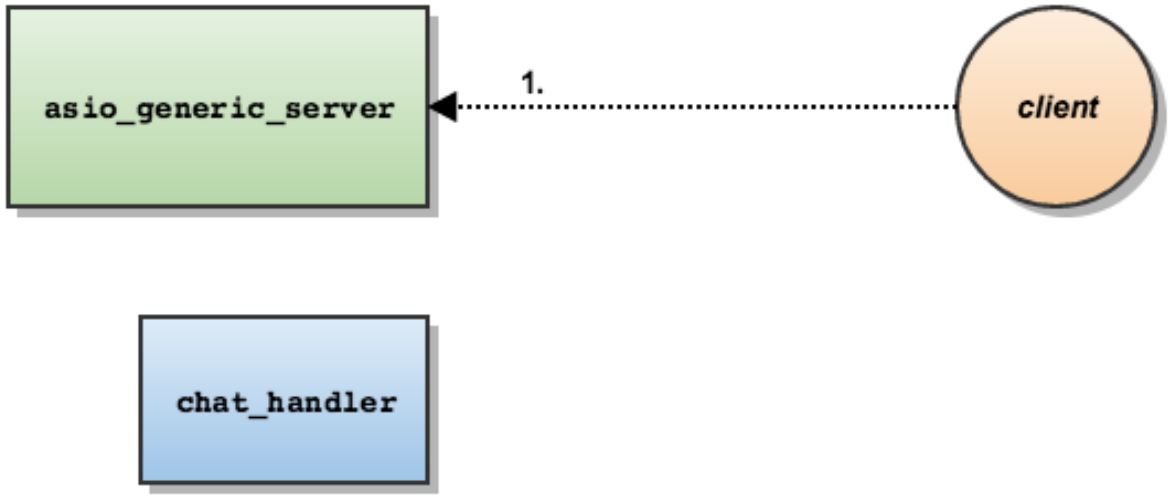
# Server

# Outline

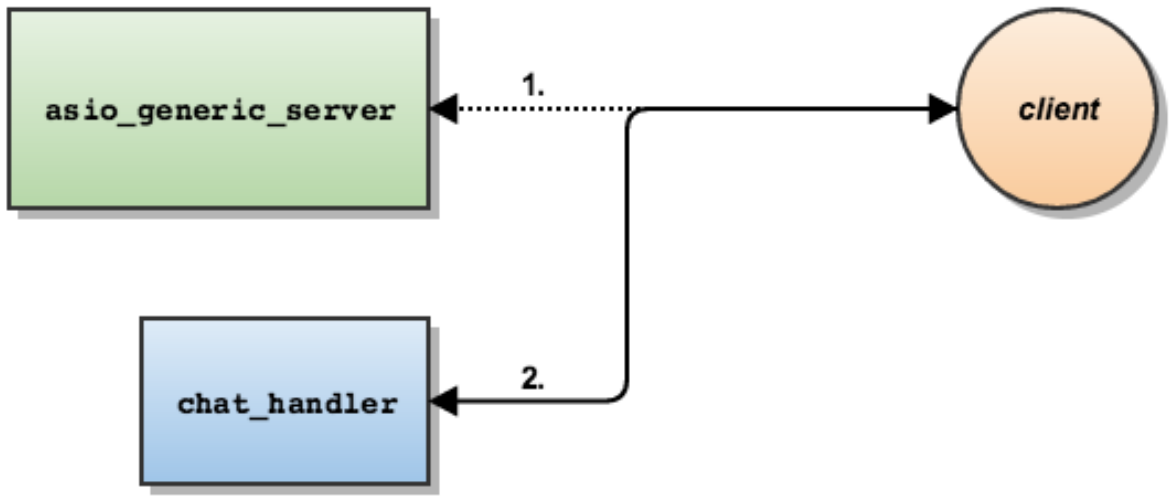
# The Chat Server



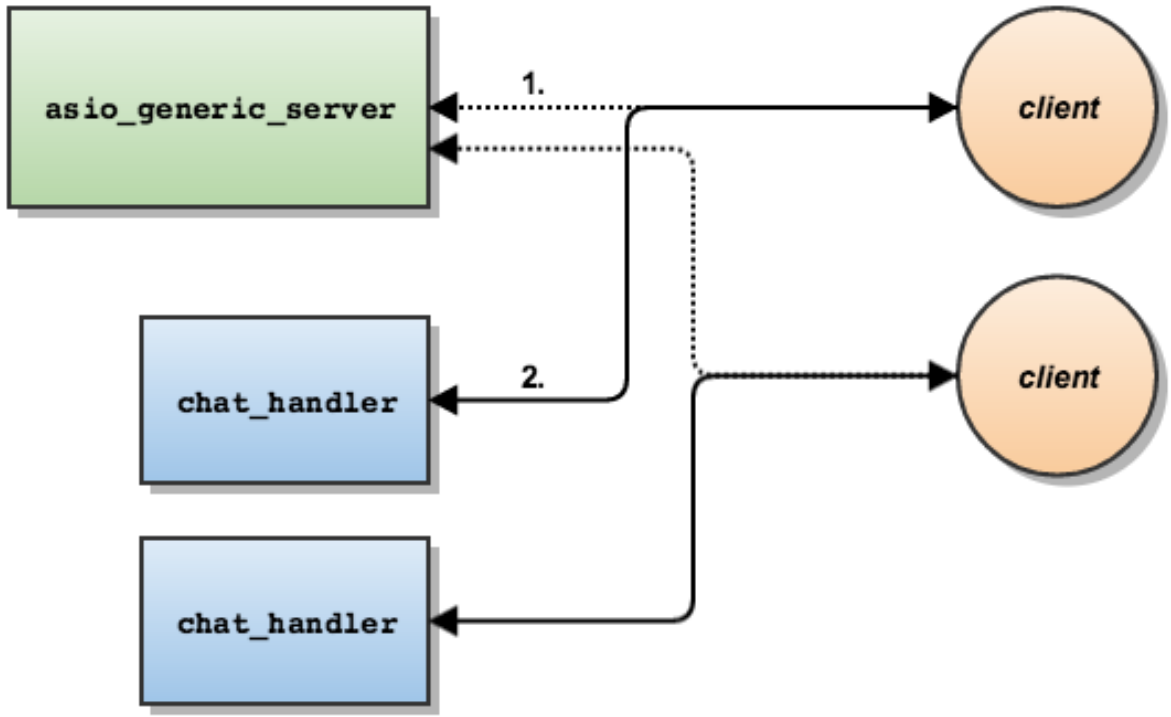
# Generic Server



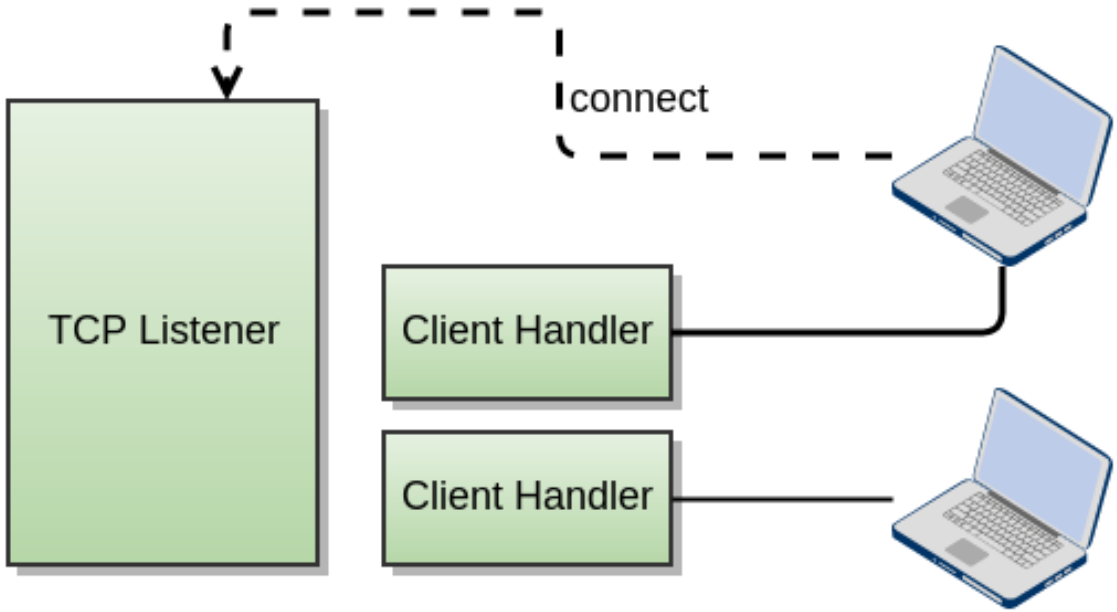
# Specific Client Handler



# Each Client has a Handler



# Generic Listener

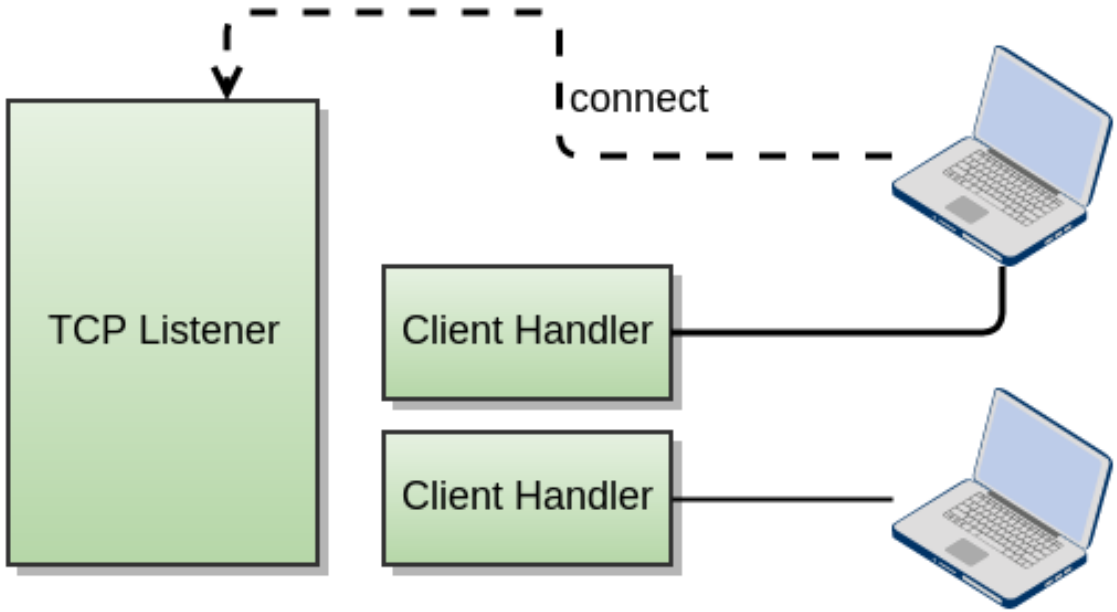



Who owns the Client Handler?



ciere.com

# Generic Listener

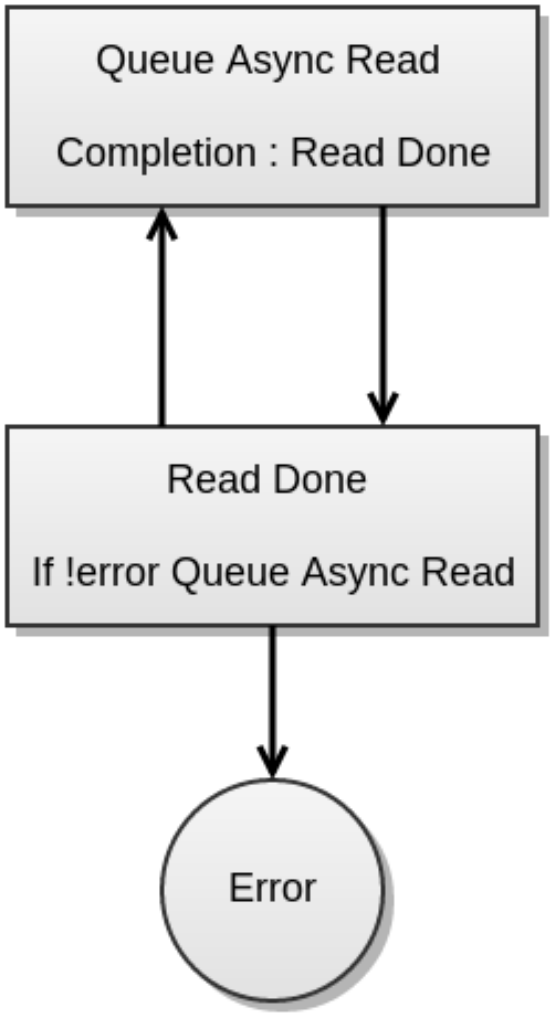



Who owns the Client Handler?



ciere.com

# Chaining Completion Handlers



# Copy a reference into the executor

```
class client_handler : std::enable_shared_from_this<client_handler>
{

public:
   client_handler(net::socket && s)
      : socket_(std::move(s))
   {}

   void start() { read_message(); }

private:
   tcp::socket socket_;
   std::string in_buffer_;

};
```

# Copy a reference into the executor

```
void read_message()
{
   net::async_read_until(socket_,
                         net::dynamic_buffer(in_buffer_),
                         '\0',
                         [me=shared_from_this()](auto ec, auto bytes_trans)
                         {
                            if(!ec)
                            {
                               me->process_message();
                            }
                         });
}
```

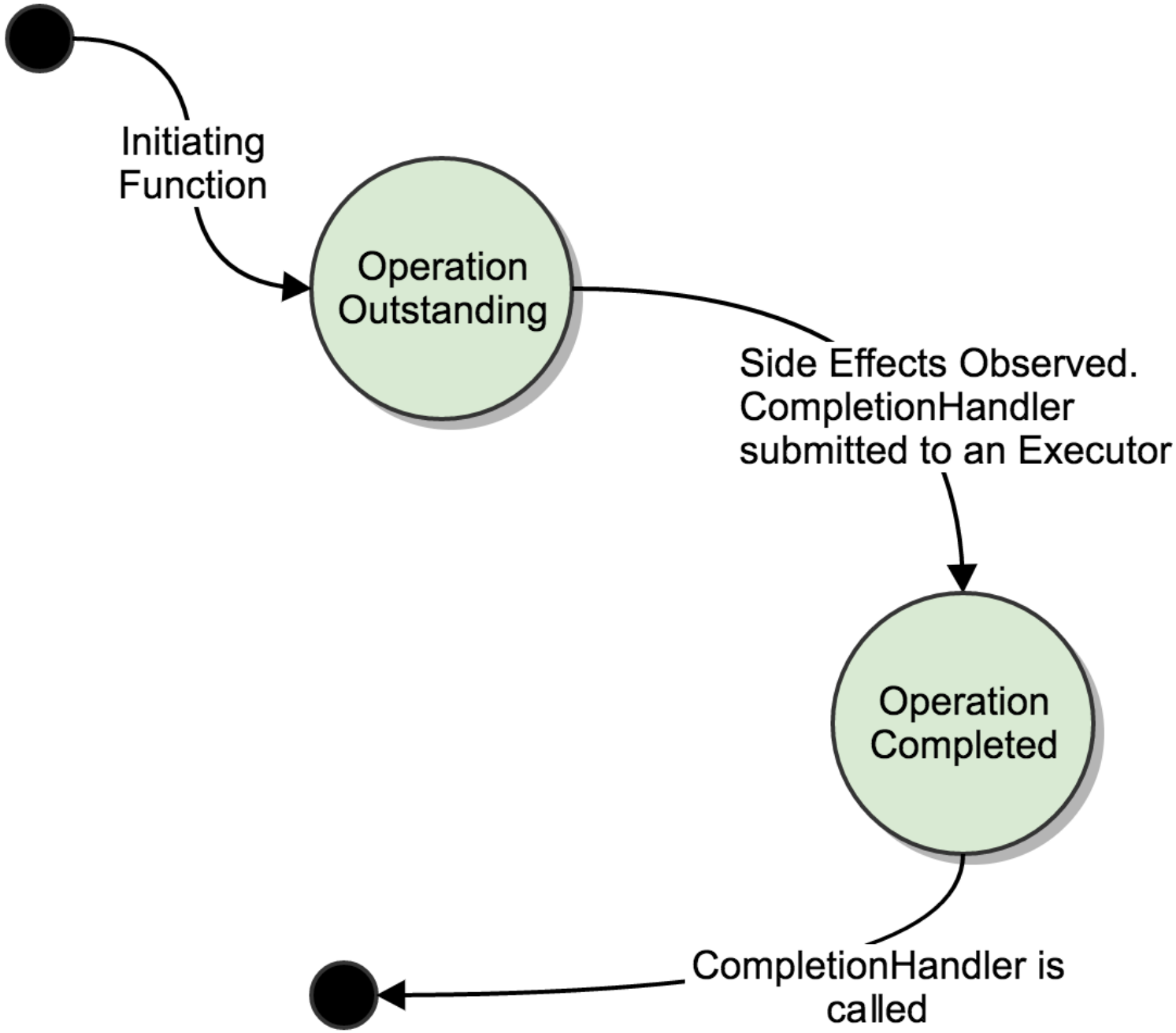
```
void process_message()
{
   // do something interesting
   // maybe push processing to a different execution_context

   // start again
   in_buffer_ = std::string{};
   read_message();
}
```

# Copy a reference into the executor

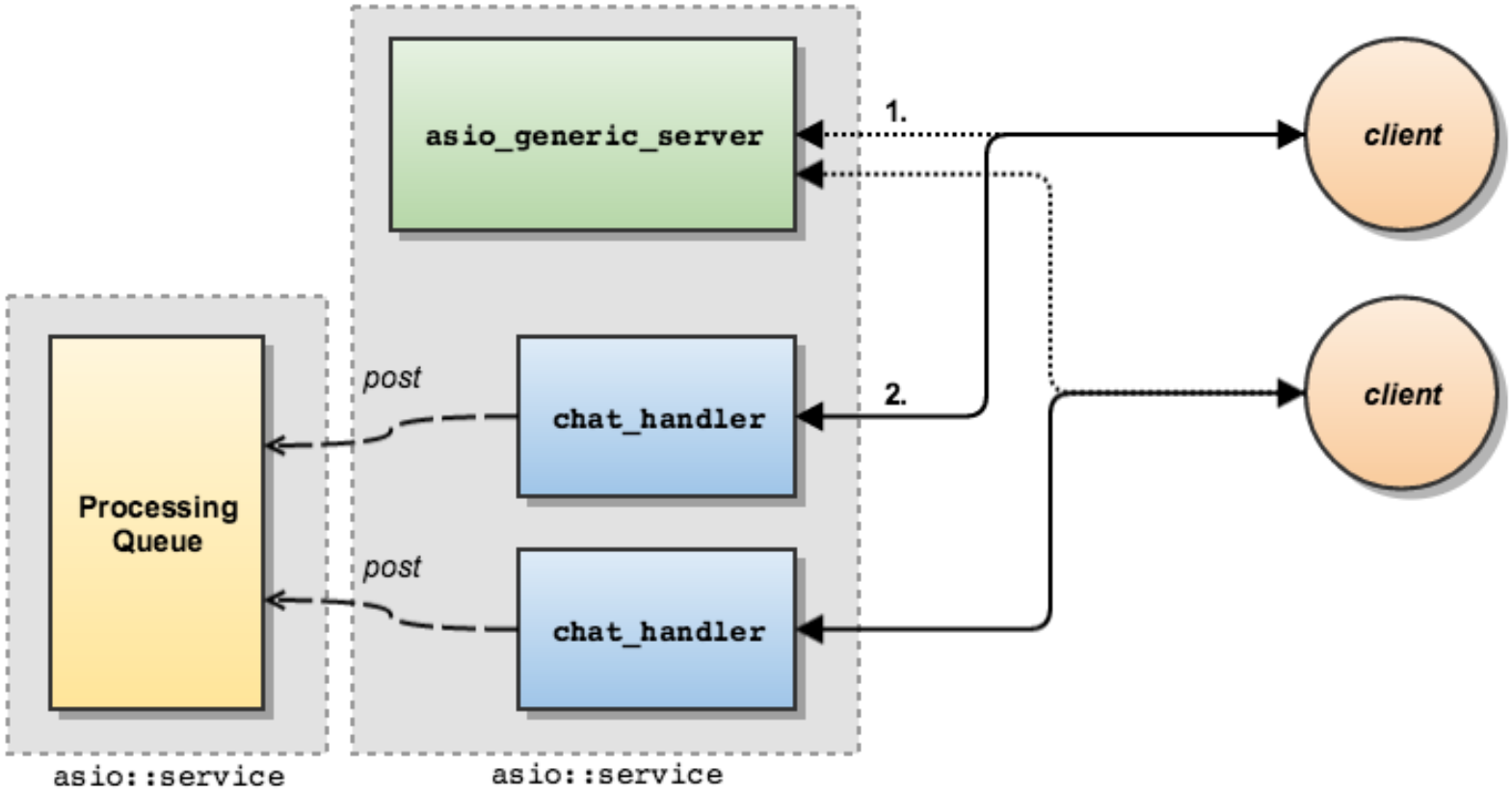
```
void read_message()
{
   net::async_read_until(socket_,
                         net::dynamic_buffer(in_buffer_),
                         '\0',
                         [me=shared_from_this()](auto ec, auto bytes_trans)
                         {
                            if(!ec)
                            {
                               me->process_message();
                            }
                         });
}
```

```
void process_message()
{
   // do something interesting
   // maybe push processing to a different execution_context

   // start again
   in_buffer_ = std::string{};
   read_message();
}
```

# Asynchronous Operation



# Decoupled Services



# More...

- Layered design!
- Use as a processing executor!
- Combine with MSM and Spirit

# Part VI

# Hands On!

# Jr. Engineering

# Win a Robot!

Using the Networking TS Implementation, submit you name to win a robot!