Full Duplex

Generalized Full Duplex Messaging

Jason Rice C++Now 2018

Overview

- Generic Network Programming
- Object Lifetime Management
- Promises
- FullDuplex Library
- Asyncronous Queueing
- Endpoint Composition

(Reads ...) ⬆

(Writes ...) ⬇

Generic Network Programming

OSI Model

Application	HTTP Websocket XMPP STUN
Presentation	SSL TLS
Session	SIP SSH H.245
Transport	TCP UDP SCTP
Network	IPv4 IPv6 IPsec
Data Link	Ethernet PPP SLIP
Physical Bits	Coax Fiber Wireless Blinky Lights

µSockets Websockets

```
uWS::Hub h;

h.onMessage([](uWS::WebSocket<uWS::SERVER> *ws,
               char *message, size_t length,
               uWS::OpCode opCode) {
    ws->send(message, length, opCode);
});
```

µSockets Websockets

```
#ifdef USE_ASIO
#include "Asio.h"
#elif !defined(__linux__) || defined(USE_LIBUV)
#include "Libuv.h"
#else
#ifndef USE_EPOLL
#define USE_EPOLL
#endif
#include "Epoll.h"
#endif
```

AMQP-CPP
RabbitMq Client

```
virtual void onData(AMQP::Connection *connection,
                    const char *data, size_t size)
{
    asio::async_write(
        socket,
        asio::const_buffer(data, size),
        [](asio::error_code const& ec, std::size_t) {
            if (ec) { /* handle error */ }
        }
    );
}
```

Generic Endpoint

```
constexpr auto my_endpoint = endpoint(
    event::init          = [](auto& state) {
        return tap([&](auto&&) {
            std::cout << "User session started: "
                      << state.session_name << '\n';
        });
    },
    event::read_message  = [](auto&) {
        return tap([](auto&& message) {
            std::cout << "Message received: "
                      << message << '\n';
        });
    },
    event::write_message = [](auto&) {
        return do_();
    }
);
```

```
struct my_state {
    tcp::socket socket;
    std::string session_name = {};
};
```

```
auto ep = endpoint_open(
    my_state{socket},
    std::queue<std::string>{},
    endpoint_compose(beast_ws_client, my_endpoint)
);
```

Object Lifetime Management

Yikes!!

```
#include <iostream>
#include <string>

auto make_handler() {
    std::string msg = "Hello, world!\n";
    return [&] { std::cout << msg; };
}

int main() {
    auto handle = make_handler();

    handle();
}
```

Yikes!!

```
#include <iostream>
#include <string>

auto make_handler() {
    std::string msg = "Hello, world!\n";
    return [&] { std::cout << msg; };
}

int main() {
    auto handle = make_handler();

    handle();
}
```

Yikes!!

```cpp
#include <iostream>
#include <string>

auto make_handler() {
    std::string msg = "Hello, world!\n";
    return [&] { std::cout << msg; };
}

int main() {
    auto handle = make_handler();

    handle();
}
```

Yikes!!

```
#include <iostream>
#include <string>

auto make_handler() {
    std::string msg = "Hello, world!\n";
    return [&] { std::cout << msg; };
}

int main() {
    auto handle = make_handler();

    handle();
}
```

Yikes!!

```
#include <iostream>
#include <string>

auto make_handler() {
    std::string msg = "Hello, world!\n";
    return [&] { std::cout << msg; };
}

int main() {
    auto handle = make_handler();

    handle();
}
```

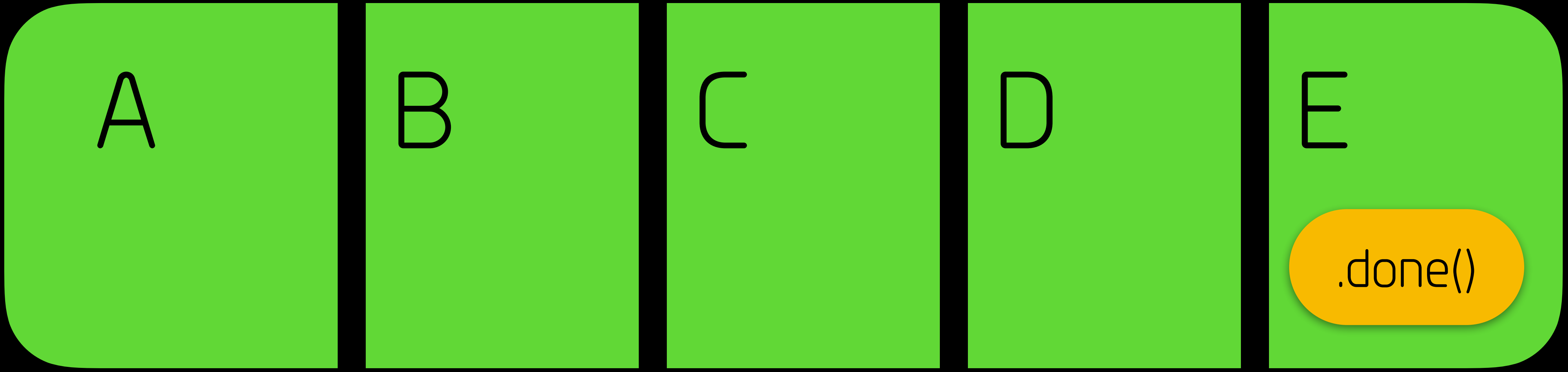
Yikes!!

```
#include <iostream>
#include <string>

auto make_handler() {
    std::string msg = "Hello, world!\n";
    return [&] { std::cout << msg; };
}

int main() {
    auto handle = make_handler();

    handle();
}
```

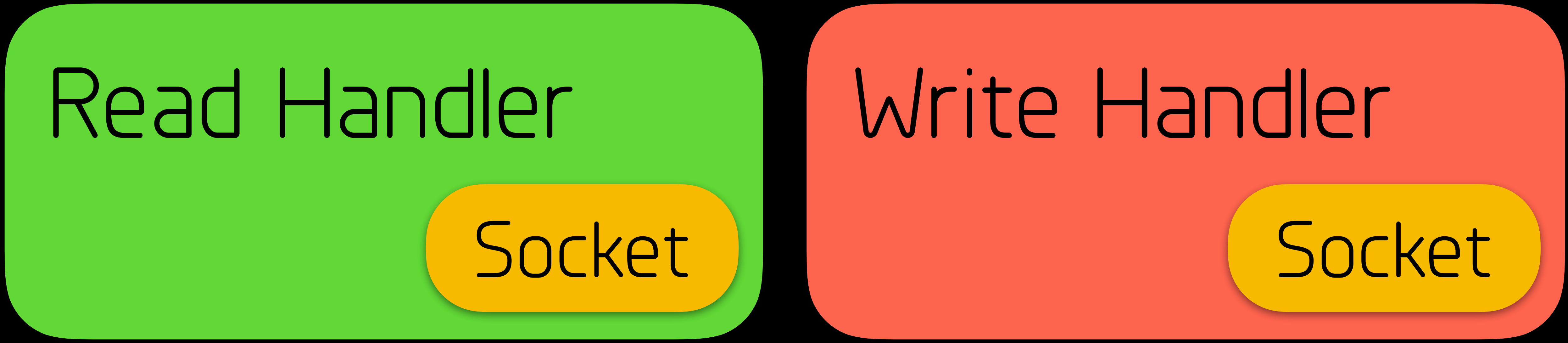
Self Ownership

```
template <typename T>
struct self_deleter {
    T value;

    void done() {
        delete this;
    }
};

int main() {
    auto me = new self_deleter<int>{5};
    me->done();
}
```

Event Series



Multiple Outstanding Handlers



Shared Ownership

```
struct writer : std::enable_shared_from_this<writer> {
    writer(tcp::socket&& socket, std::string message)
        : socket(std::move(socket))
        , message(std::move(message))
    { }

    void keep_writing() {
        asio::async_write(socket,
                          asio::buffer(message, message.size()),
            [this, _(shared_from_this())](std::error_code error,
                                          std::size_t)
            {
                if (not error) keep_writing();
            });
    }

    tcp::socket socket;
    std::string message;
};
```

Shared Ownership

```
struct writer : std::enable_shared_from_this<writer> {
    writer(tcp::socket&& socket, std::string message)
        : socket(std::move(socket))
        , message(std::move(message))
    { }

    void keep_writing() {
        asio::async_write(socket,
                          asio::buffer(message, message.size()),
            [this, _(shared_from_this())](std::error_code error,
                                          std::size_t)
            {
                if (not error) keep_writing();
            });
    }

    tcp::socket socket;
    std::string message;
};
```

Shared Ownership

```
struct writer : std::enable_shared_from_this<writer> {
    writer(tcp::socket&& socket, std::string message)
        : socket(std::move(socket))
        , message(std::move(message))
    { }

    void keep_writing() {
        asio::async_write(socket,
                          asio::buffer(message, message.size()),
            [this, _(shared_from_this())](std::error_code error,
                                          std::size_t)
            {
                if (not error) keep_writing();
            });
    }

    tcp::socket socket;
    std::string message;
};
```

Promises

Dependent

```
[](auto x) { return x + x; }
```

Function Composition

```cpp
auto foo = [](auto x) { return x + x; };
auto bar = [](auto x) { return x - 1; };
auto baz = [](auto x) { std::cout << x << '\n'; return x; };
```

Function Composition

```
baz(
bar(
foo(
 42
)));
```

Function Composition

```
baz(
bar(
foo(
 42
)));
```

Function Composition

```
baz(
bar(
foo(
 42
)));
```

Function Composition

```
baz(
bar(
foo(
  42
)));
```

Function Composition

```
baz(
bar(
foo(
 42
)));
```

```
# ./a.out
83
```

Callbacks

```
auto foo = [](auto resolve) { return [=](auto x) {
    resolve(x + x);
};};
```

Callbacks

```
auto bar = [](auto resolve) { return [=](auto x) {
    if (x < 50) {
        resolve(x);
    } else {
        resolve(error{"out of range"});
    }
};};
```

Callbacks

```
auto baz = [](auto resolve) { return [=](auto x) {
    std::cout << x << '\n';
    resolve(x);
};};
```

Callbacks

```
foo(
bar(
baz(
 noop
))) (5);
```

Callbacks

```
foo(
bar(
baz(
 noop
))) (5);
```

Callbacks

```
foo(
bar(
baz(
 noop
))) (5);
```

Callbacks

```
foo(
bar(
baz(
 noop
))) (5);
```

Callbacks

```
foo(
bar(
baz(
 noop
))) (5);
```

Callbacks

```
foo(
bar(
baz(
 noop
))) (5);

foo(
bar(
baz(
 noop
))) (42);
```

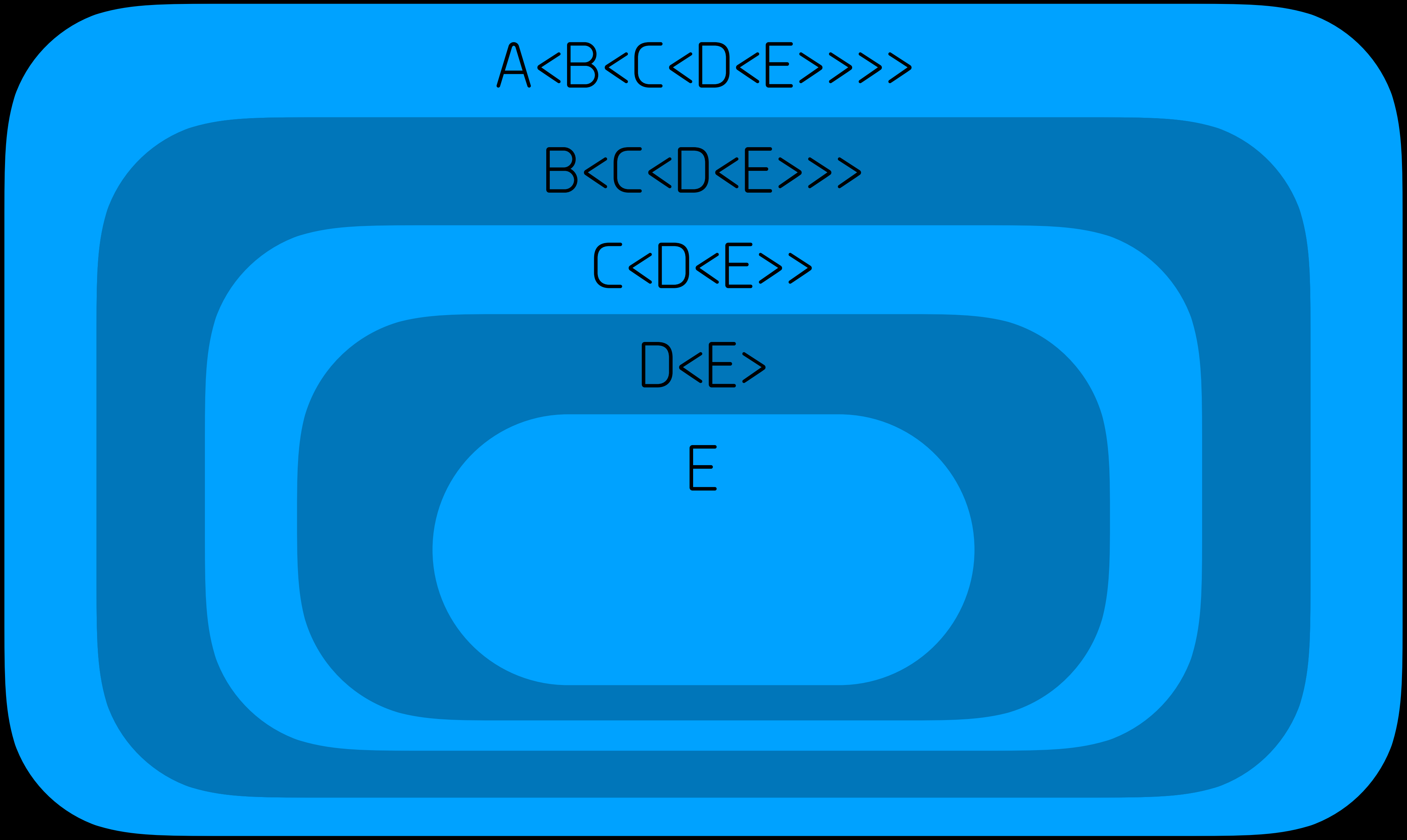
```
# ./a.out
10
error("out of range")
```

Promise

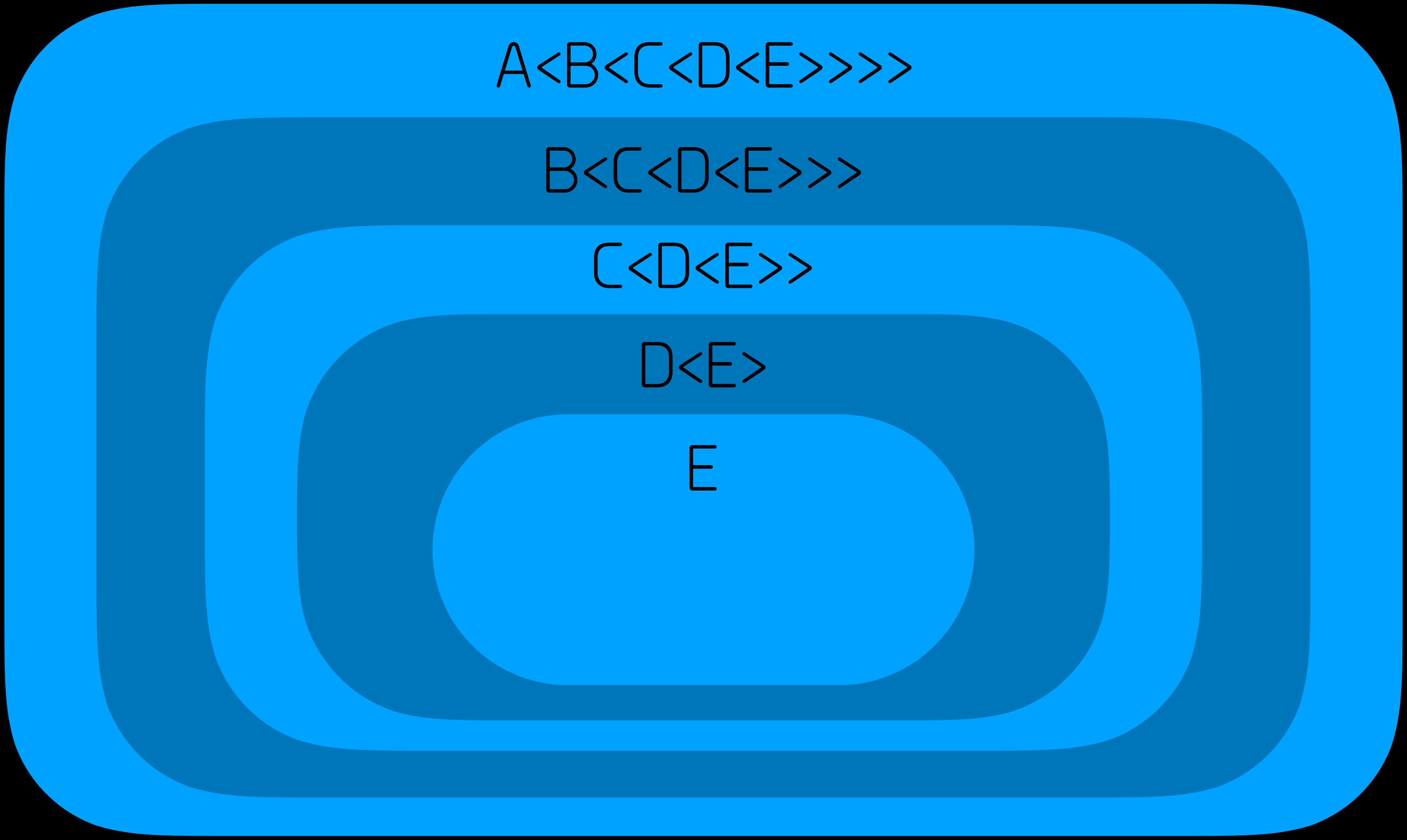
Promise

```
auto f =
    foo(
    bar(
    baz(
     noop
    )));

auto fg = uhhh(f, g);
```

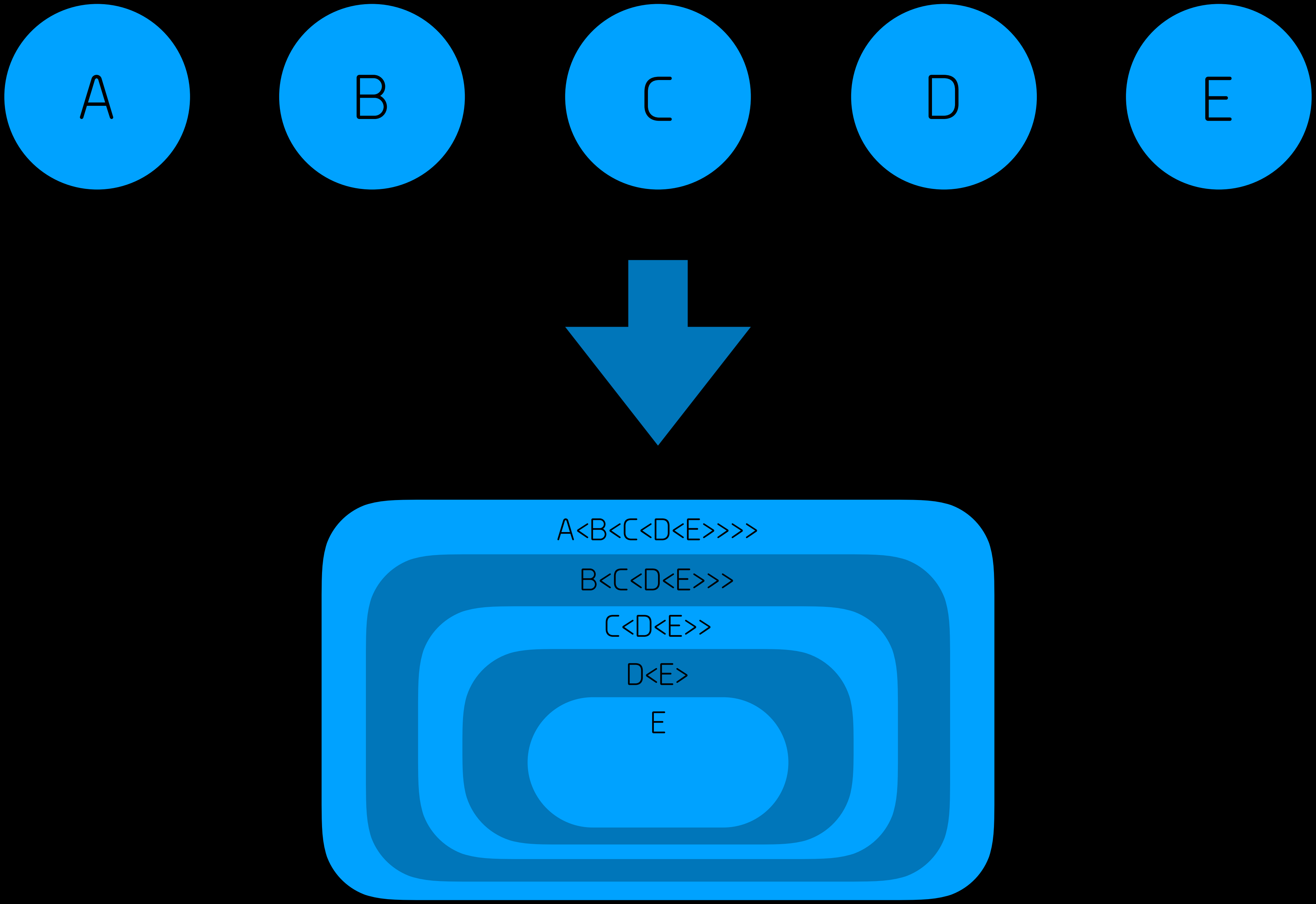
Promise



Promise



Promise



Promise

```
auto promise_f = do_(
    promise(foo),
    promise(bar),
    promise(baz)
);

auto promise_fg = do_(promise_f, promise_g)

final_promise(promise_fg)(42);
```

Promise

```
promise([](auto& resolve, auto x) {
    if (x < 50) {
        resolve(x);
    } else {
        resolve(error{"out of range"});
    }
});
```

Promise Monadic Interface p0650R0-ish

- do_
- promise
- map
- tap
- map_error
- map_either
- map_any
- catch_error

Promise

```
hana::is_a<promise_tag>(map(hana::id))
```

Promise

```
map(hana::id)(42) == promise_lift(42)
```

Promise

```
chain(p1, p2) == concat(p1, p2)
```

Promise

```
hana::chain(
    promise_lift(int{5}),
    [](int x) {
        return promise_lift(x * x);
    }
)
```

Promise

```
hana::chain(
    promise_lift(5),
    promise([](auto& resolve, auto x) {
        resolve(x * x);
    })
)
```

Promise

```
hana::chain(
    hana::chain(
        promise_lift(5),
        [](int x) { return promise_lift(x * x); }
    ),
    [](int x) { return promise_lift(x * x); }
)
```

Do "Notation"

```
do_(
    connect,
    handshake_request,
    handshake_response,
    parse_auth_token,
    keep_reading,
    catch_error(shutdown)
)
```

Promise Execution

```
run_async(
    connect,
    handshake_request,
    handshake_response,
    parse_auth_token,
    keep_reading,
    catch_error(shutdown)
);
```

Promise Tail

```
template <typename Input>
void operator()(Input const&) {
    static_assert(
        not hana::is_an<error_tag, Input>,
        "Unhandled Promise Error!"
    );
    delete self;
}
```

Promise Loop

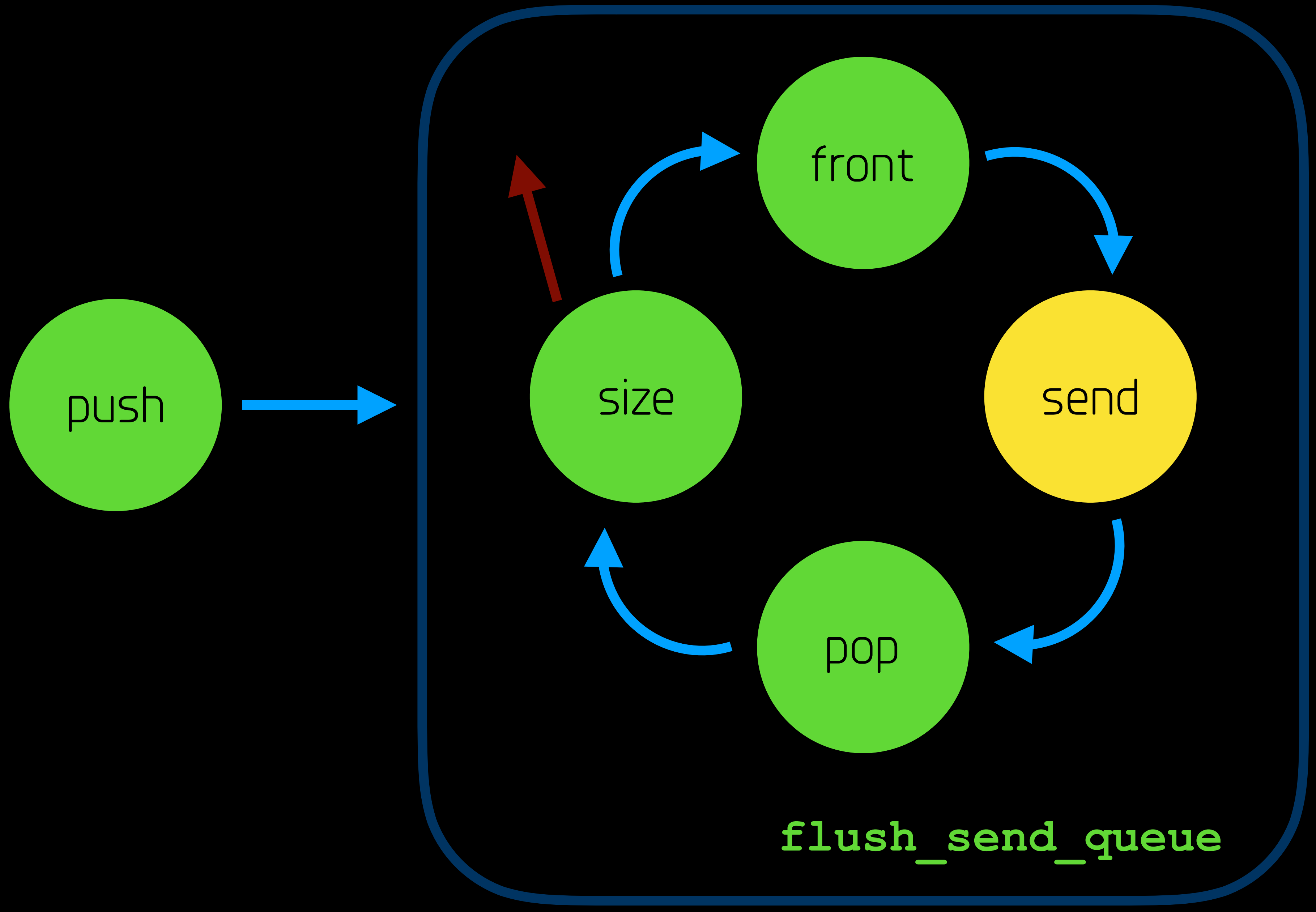
```
void keep_reading() {
    run_async_loop(
        terminate_if_stopped(),
        endpoint.read_message(state),
        error_catcher()
    );
}
```

Promise Loop

```
template <typename Input>
void operator()(Input&& input) {
    static_assert(
        not hana::is_an<error_tag, Input>,
        "Unhandled Promise Error!"
    );

    if constexpr(hana::is_a<terminate, Input>) {
        delete self;
    }
    else {
        // start the promise over again
        self->promise_sum(std::forward<Input>(input));
    }
}
```

Asyncronous Queueing

Gleaming the Queue



Push

```
run_async(
    promise_lift(message),
    send_queue.push(),
    tap([this](void_input_t) {
        flush_send_queue();
    }),
    error_catcher()
);
```

Flush

```
run_async_loop(
    terminate_if_done,
    terminate_if_stopped(),
    send_queue.front(),
    terminate_if_stopped(),
    endpoint.write_message(state),
    send_queue.pop(),
    error_catcher()
);
```

Front (std::queue)

```
template <typename Queue>
constexpr decltype(auto)
async_queue<Queue>::front() {
    return map([this](auto&&) noexcept {
        assert(queue.size() > 0);
        return queue.front();
    });
}
```

Endpoints

Endpoint Event

```
constexpr auto connect = [](auto& self) {
    return promise([&](auto& resolve, auto&&) {
        auto& state = self.state();
        state.socket().async_connect(state.endpoint,
            [&](auto error) {
                (not error) ? resolve(self)
                            : resolve(make_error(error));
            });
    });
};
```

Endpoint "Classes"

```
constexpr auto acceptor   = endpoint(event::init = accept);
constexpr auto connector  = endpoint(event::init = connect);

constexpr auto message = endpoint(
    event::read_message  = read_message,
    event::write_message = write_message
);
```

Endpoint Compose

```
endpoint_compose(asio_tcp::connector,
                 asio_tcp::message,
                 my_message);


endpoint_compose(asio_tcp::connector,
                 beast_ws::message,
                 my_message);
```

Endpoint Compose-ish

```
auto endpoint_compose(auto T, auto U) {
    return endoint(
        init          = do_(T.init, U.init]),
        read_message  = do_(T.read_message,  U.read_message),
        write_message = do_(U.write_message, T.write_message));
}
```

Endpoint Compose-ish

```
auto endpoint_compose(auto T, auto U) {
    return endoint(
        init          = do_(T.init, U.init]),
        read_message  = do_(T.read_message,  U.read_message),
        write_message = do_(U.write_message, T.write_message));
}
```

TCP Messaging

```
constexpr auto read_message = [](auto& self) {
    using Self = decltype(self);
    return do_(
        promise(read_length_fn<Self>(self)),
        promise(read_body_fn<Self>{self})
    );
};
```

TCP Messaging

```
template <typename Self>
struct read_length_fn {
    template <typename Resolve, typename Input>
    auto operator()(Resolve& resolve, Input&&) {
        asio::async_read(self.state().socket(),
                         asio::buffer(buffer, 4),
                         [&](auto error, size_t) {
            uint32_t length = buffer[0] << 24
                            | buffer[1] << 16
                            | buffer[2] << 8
                            | buffer[3];
            (not error) ? resolve(length)
                        : resolve(make_error(error));
        });
    }

    template <typename S>
    explicit read_length_fn(S& s)
        : self(s)
        , buffer()
    { }

    Self& self;
    unsigned char buffer[4];
};
```

TCP Messaging

```
template <typename Self>
struct read_body_fn {
    template <typename Resolve>
    auto operator()(Resolve& resolve, uint32_t length) {
        body.resize(length);
        asio::async_read(self.state().socket(),
                         asio::buffer(body, length),
                         [&](auto error, size_t) {
            (not error) ? resolve(body)
                        : resolve(make_error(error));
        });
    }

    explicit read_body_fn(Self& s)
        : self(s)
        , body()
    { }

    Self& self;
    std::string body;
};
```

TCP Messaging

```
constexpr auto write_message = [](auto& self) {
    tcp::socket& socket = self.state().socket();

    return do_(
        promise(write_length_fn{socket}),
        promise([&](auto& resolve, auto& message) {
            asio::async_write(
                socket,
                asio::buffer(message, message.size()),
                [&](auto error, size_t) {
                    (not error) ? resolve(message)
                                : resolve(make_error(error));
                });
        })
    );
};
```

Websocket Messaging

```
template <typename Self>
struct read_message_fn {
    template <typename Resolve, typename Input>
    void operator()(Resolve& resolve, Input&&) {
        self.state().ws.async_read(buffer, [&](auto error, size_t) {
            auto mut_buf = buffer.data();
            std::string_view buffer_view(
                static_cast<char const*>(mut_buf.data()),
                mut_buf.size());
            body.resize(buffer_view.size());
            std::copy(buffer_view.begin(),
                      buffer_view.end(),
                      body.begin()); // :(
            buffer.consume(buffer.size());
            (not error) ? resolve(body)
                        : resolve(make_error(error));
        });
    }

    explicit read_message_fn(Self& s)
        : self(s)
        , body()
        , buffer()
    { }

    Self& self;
    std::string body;
    boost::beast::flat_buffer buffer;
};
```

Websocket Messaging

```
// Networking TS
template<typename AsyncReadStream,
         typename DynamicBuffer,
         typename ReadHandler>
auto async_read(AsyncReadStream& s,
                DynamicBuffer&& buffers,
                ReadHandler&& handler);


// Beast Websocket (member function)
template<class DynamicBuffer,
         class ReadHandler>
auto async_read(DynamicBuffer& buffer,
                ReadHandler&& handler);
```

Websocket Messaging

```
constexpr auto write_message = [](auto& self) {
    return promise([&](auto& resolve, auto& message) {
        self.state().ws.async_write(
            asio::buffer(message, message.size()),
            [&](auto error, size_t) {
                (not error) ? resolve(message)
                            : resolve(make_error(error));
            }
        );
    });
};
```

User Level Messaging

```
endpoint(
    event::read_message = [](auto& self) {
        return promise([&](auto& resolve, auto const& msg) {
            if (msg == "terminate") {
                resolve(full_duplex::terminate{});
            }
            else {
                // echo the message
                self.send_message(msg);
                resolve(msg);
            }
        });
    },
    event::error = [](auto&) {
        return tap([](auto& error) {
            std::cout << "LISTENER ERROR: "
                      << error.message() << " \n";
        });
    }
)
```

```
full_duplex::send_message(
        sender,
        std::string("terminate"));
```

- https://github.com/ricejasonf/cppnow18_full_duplex
- https://github.com/ricejasonf/full_duplex
- https://github.com/boostorg/hana
- https://github.com/boostorg/beast
- https://github.com/boostorg/asio
- https://github.com/CopernicaMarketingSoftware/AMQP-CPP
- https://github.com/uNetworking/uWebSockets
- https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0650r0.pdf