
Core C++ 2024
Evolving C++ Networking With Senders & Receivers
Part 1
Robert Leahy

How do you write good code?

Split it up

How do you split code up?

Call a function

LSEG

How do you call a function?


foo(x)

How do you call a function?

Overthinking something we do everyday

1. Prepare/synthesize the arguments
2. Pass the prepared/synthesized values
3. Allow the function to run
4. Collect the returned value or receive the emitted exception

What if you want to split this up?

Breaking up function calls

Further separation of concerns

Preparation/synthesis of arguments

Currying separates part (partial application) or all (full application) of this process from the point at which those arguments are actually provided to the function

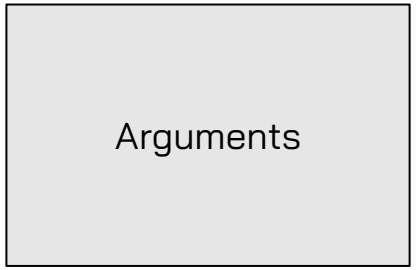
Regular functions have no answer for everything else

Because regular functions are synchronous once the caller commits to passing a certain concrete set of arguments to the function nothing else can be separated out

- Arguments are passed on the stack, so can't split that out while continuing to use the stack
- Called function runs on the CPU, so can't split that out while continuing to use the CPU
- Return occurs to the point of call, so can't split that out because the program would move on from the point of call

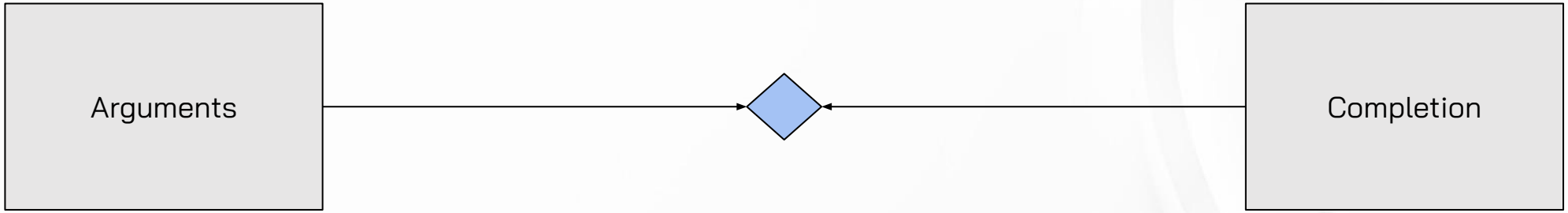
Modeling a Function Call

Modeling a Function Call

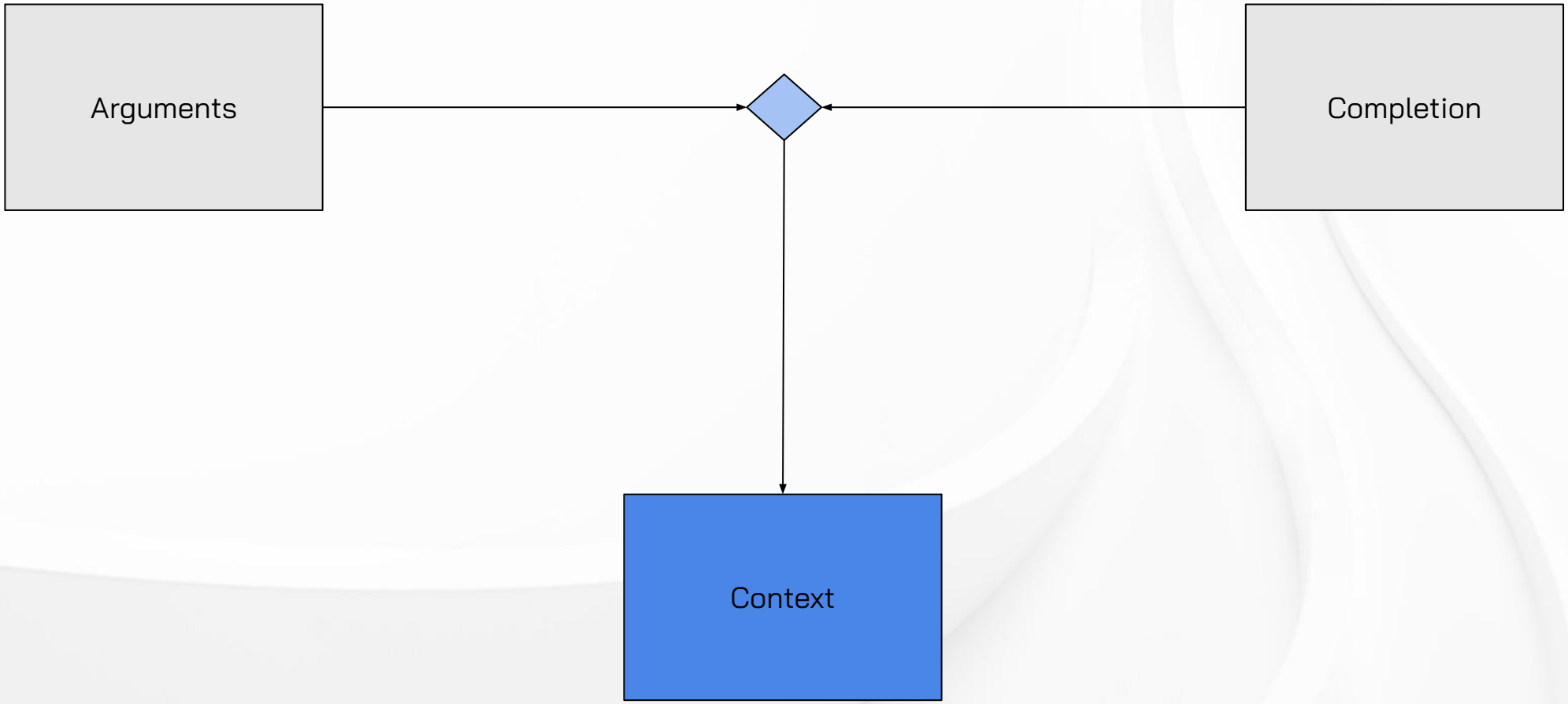


Modeling a Function Call

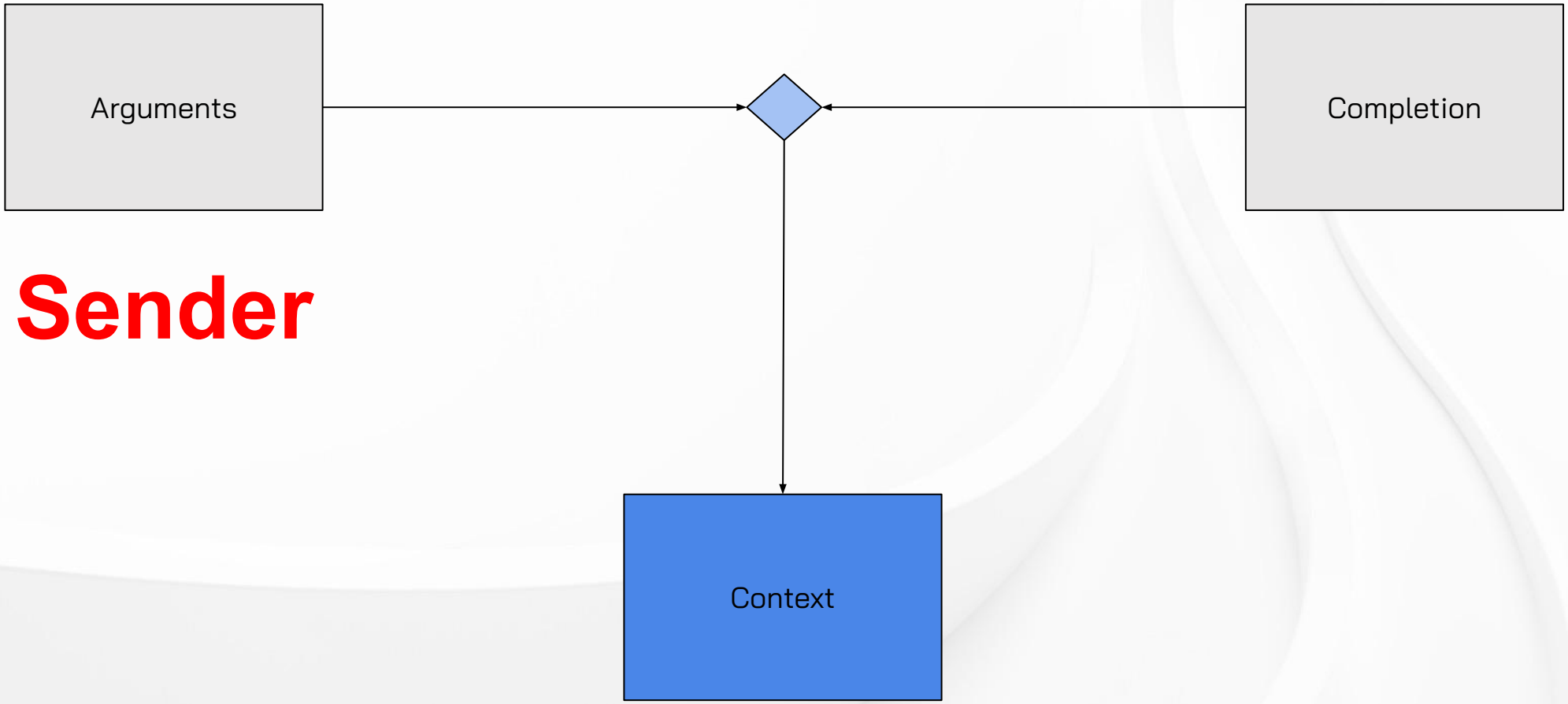
Modeling a Function Call



Modeling a Function Call

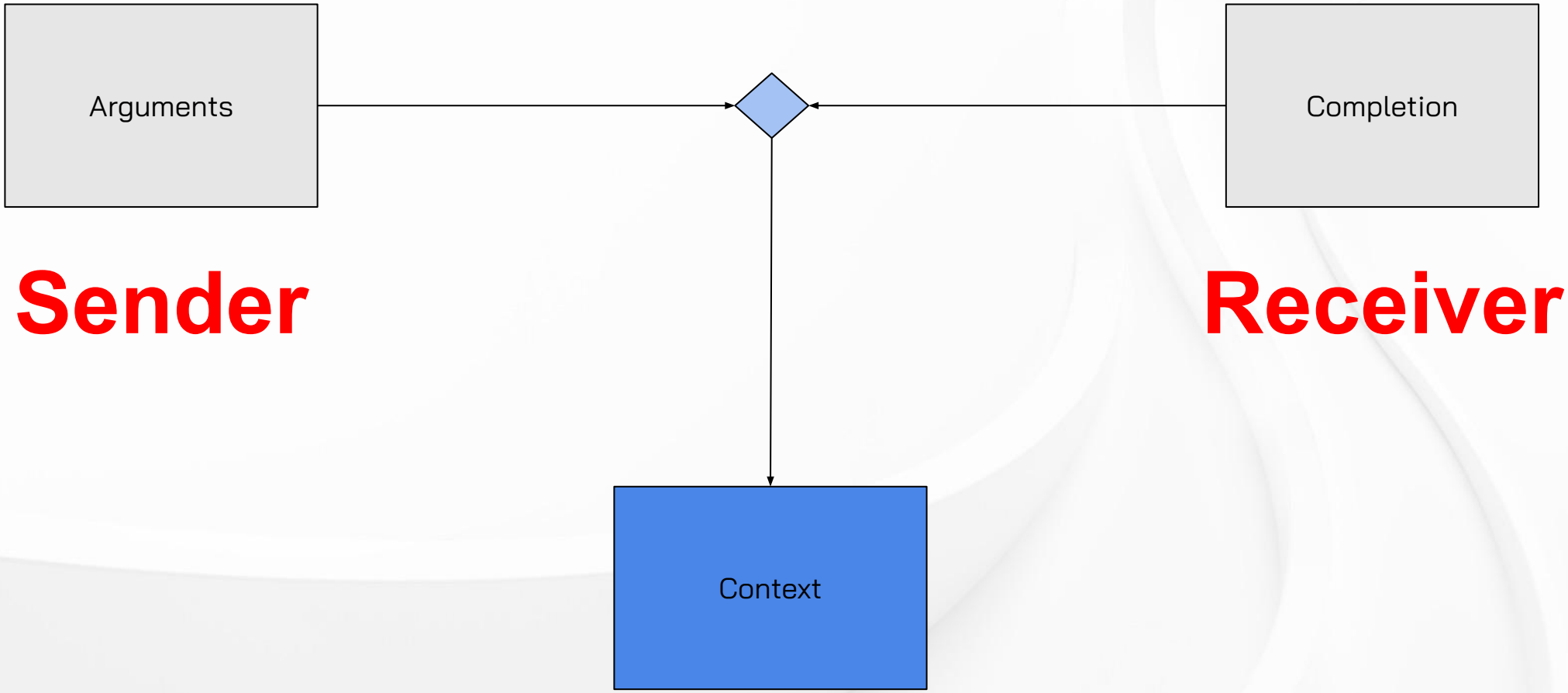


Modeling a Function Call

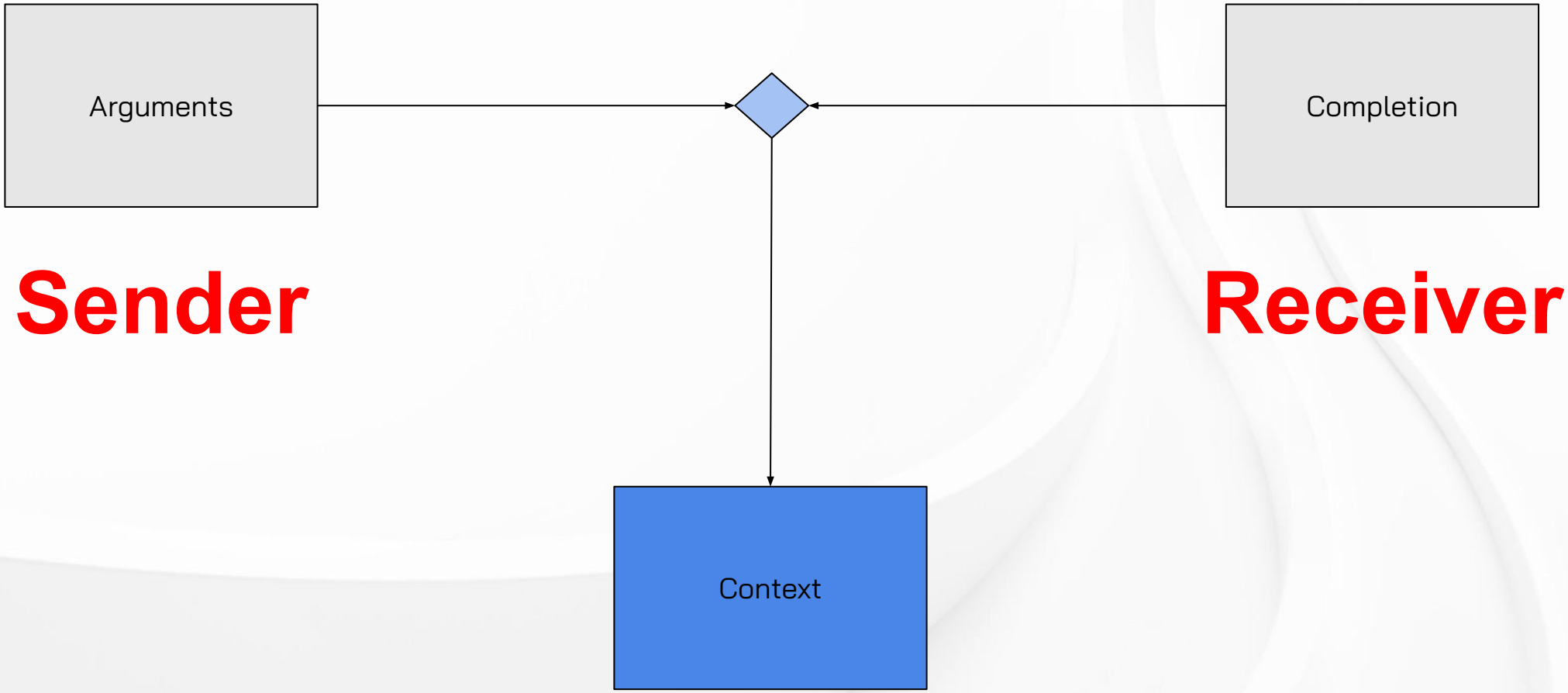


Sender

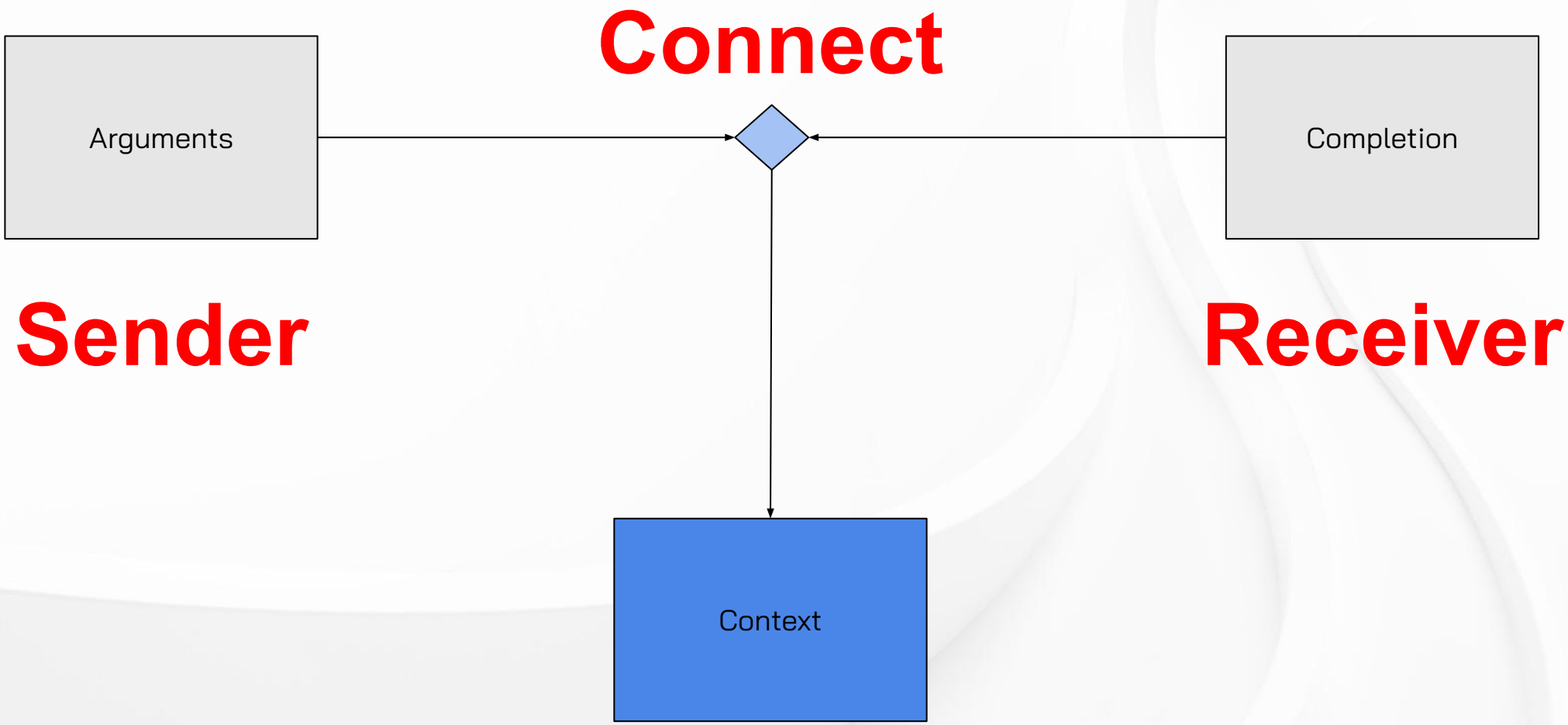
Modeling a Function Call



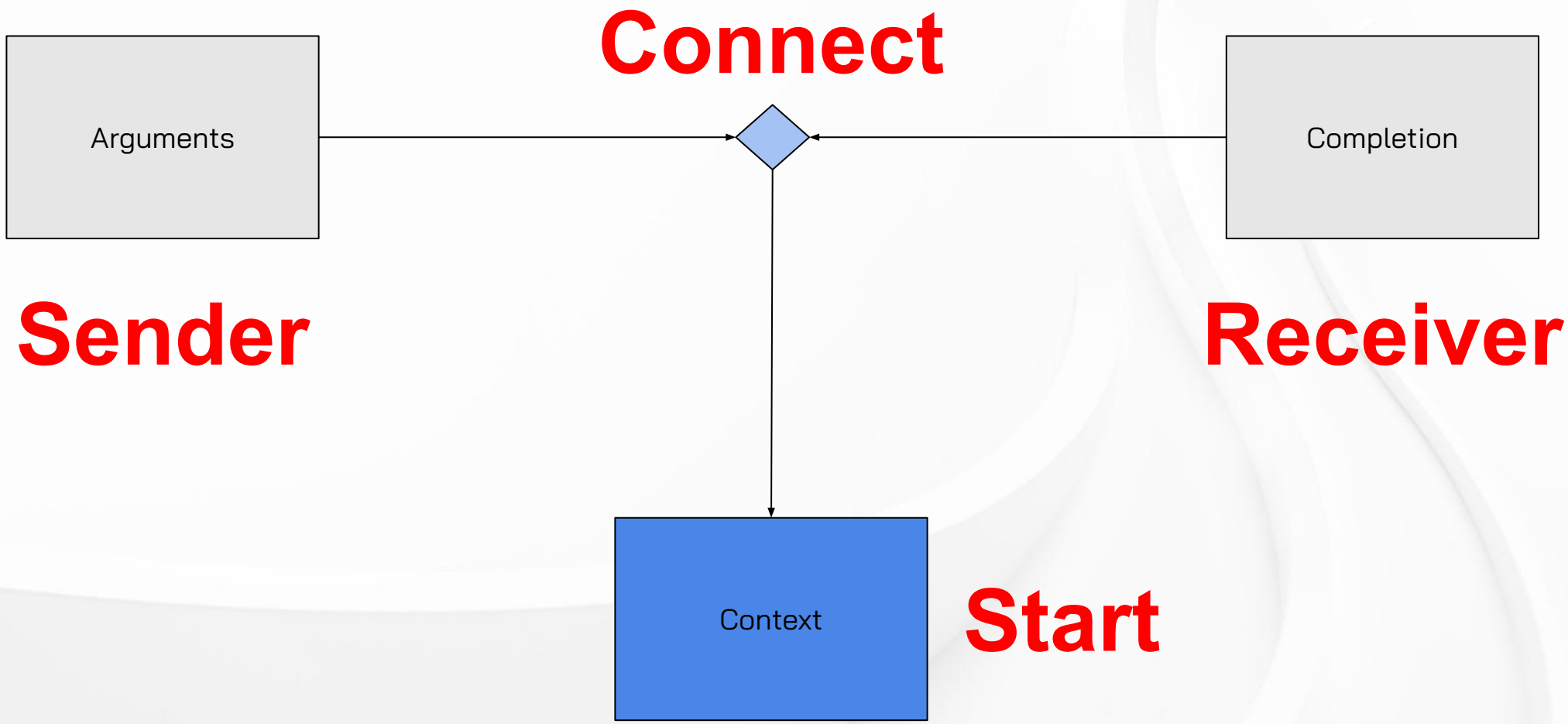
Modeling a Function Call



Modeling a Function Call



Modeling a Function Call



Synchronous Example

The lambda simply returns `5`.

The code which calls the lambda stores the value returned thereby in `i`.

Before the lambda is called `!i` is obviously `true`.

```
std::optional<int> i;
const auto f = []() noexcept { return 5; };
//  !i is true
i = f();
//  i == 5 is true
```

Asynchronous Example

The code on the preceding slide is rewritten asynchronously using senders & receivers.

Note that unlike in the synchronous example the call can be fully setup (by `connect`) without its side effects occurring (this doesn't occur until `start`).

```
std::optional<int> i;
const auto sender = std::execution::just(5);
const struct {
  using receiver_concept = std::execution::receiver_t;
  void set_value(const int i) && noexcept {
    this->i = i;
  }
  std::optional<int>& i;
} receiver{i};
auto op = std::execution::connect(sender, receiver);
//  !i is true
std::execution::start(op);
//  i == 5 is true
```

Asynchronous Example

Creation of the sender crystallizes what the asynchronous operation will do, in the synchronous domain it's the same as selecting the function to call and deciding on its arguments.

In this case we don't actually select any arguments so this is less interesting than it might otherwise be.

```
std::optional<int> i;
const auto sender = std::execution::just(5);
const struct {
  using receiver_concept = std::execution::receiver_t;
  void set_value(const int i) && noexcept {
    this->i = i;
  }
  std::optional<int>& i;
} receiver{i};
auto op = std::execution::connect(sender, receiver);
//  !i is true
std::execution::start(op);
//  i == 5 is true
```

LSEG

Asynchronous Example

The receiver determines what happens when the asynchronous operation finishes, in the synchronous domain this is the same as the action that occurs when the function occurs.

Note that in the synchronous example we assigned the returned value to `i` whereas here we assign the value provided to `set_value` to `i`, thus codifying the isomorphism described above.

```
std::optional<int> i;
const auto sender = std::execution::just(5);
const struct {
  using receiver_concept = std::execution::receiver_t;
  void set_value(const int i) && noexcept {
    this->i = i;
  }
  std::optional<int>& i;
} receiver{i};
auto op = std::execution::connect(sender, receiver);
//  !i is true
std::execution::start(op);
//  i == 5 is true
```

LSEG

Asynchronous Example

The function and its arguments (the sender) and the point to "return" to (the receiver) are brought together.

op is the "operation state," encapsulates the operation, and is isomorphic to the stack frame of a synchronous operation.

Note that unlike in the synchronous domain the function isn't running yet, it's just been fully setup.

Also note that operation states are immovable, once we collapse the prvalue we can't move the operation state further.

```
std::optional<int> i;
const auto sender = std::execution::just(5);
const struct {
  using receiver_concept = std::execution::receiver_t;
  void set_value(const int i) && noexcept {
    this->i = i;
  }
  std::optional<int>& i;
} receiver{i};
auto op = std::execution::connect(sender, receiver);
//  !i is true
std::execution::start(op);
//  i == 5 is true
```

LSEG

Asynchronous Example

Starting the operation state actually begins the operation, which in this case completes inline.

Note that this is guaranteed not to throw exceptions, and once it's undertaken the caller has passed the point of no return.

Just as in the synchronous domain you can't "un-call" a function, instead needing to wait for the return, in the asynchronous domain you can't "un-start" an asynchronous operation, instead you need to ensure the operation state remains valid until the operation completes.

```
std::optional<int> i;
const auto sender = std::execution::just(5);
const struct {
  using receiver_concept = std::execution::receiver_t;
  void set_value(const int i) && noexcept {
    this->i = i;
  }
  std::optional<int>& i;
} receiver{i};
auto op = std::execution::connect(sender, receiver);
//  !i is true
std::execution::start(op);
//  i == 5 is true
```

LSEG

The Receiver Contract

- Unstarted operation state may be destroyed at any time
  - No completion signal sent to receiver
- Started operation state may not simply be destroyed
  - Lifetime must persist at least until completion signal sent to receiver
  - Exactly one completion signal sent to receiver

The Receiver Contract

- Unstarted operation state may be destroyed at any time
  - No completion signal sent to receiver
- Started operation state may not simply be destroyed
  - Lifetime must persist at least until completion signal sent to receiver
  - Exactly one completion signal sent to receiver

`set_value` is a completion signal which transmits success

The Receiver Contract

- Unstarted operation state may be destroyed at any time
  - No completion signal sent to receiver
- Started operation state may not simply be destroyed
  - Lifetime must persist at least until completion signal sent to receiver
  - Exactly one completion signal sent to receiver

`set_value` is a completion signal which transmits success

What about other completion conditions?

Synchronous Example

This example adds a filtering step to the previous example: Only integers greater than 10 are allowed.

```
const auto f1 = []() noexcept { return 5; };
const auto f2 = [](auto f) {
  auto i = f();
  if (i > 10) {
    return i;
  }
  throw std::runtime_error("Less than or equal to 10");
};
//  Throws
(void)f2(f1);
```

LSEG

Asynchronous Example

Same result, similar code structure, except asynchronous.

```
std::optional<std::variant<int, std::exception_ptr>> o;
const auto sender = std::execution::just(5);
const auto adaptor = std::execution::then([](const int i) {
  if (i > 10) {
    return i;
  }
  throw std::runtime_error("Less than or equal to 10");
});
const struct {
  using receiver_concept = std::execution::receiver_t;
  void set_value(const int i) && noexcept {
    o.emplace(i);
  }
  void set_error(std::exception_ptr ex) && noexcept {
    o.emplace(std::move(ex));
  }
  std::optional<std::variant<int, std::exception_ptr>>& o;
} receiver{o};
auto op = std::execution::connect(sender | adaptor, receiver);
//  !o is true
std::execution::start(op);
//  o holds a variant whose active alternative is
//  std::exception_ptr
```

LSEG

Asynchronous Example

`then` allows the values yielded by a predecessor asynchronous operation to be transformed.

```
std::optional<std::variant<int, std::exception_ptr>> o;
const auto sender = std::execution::just(5);
const auto adaptor = std::execution::then([](const int i) {
  if (i > 10) {
    return i;
  }
  throw std::runtime_error("Less than or equal to 10");
});
const struct {
  using receiver_concept = std::execution::receiver_t;
  void set_value(const int i) && noexcept {
    o.emplace(i);
  }
  void set_error(std::exception_ptr ex) && noexcept {
    o.emplace(std::move(ex));
  }
  std::optional<std::variant<int, std::exception_ptr>>& o;
} receiver{o};
auto op = std::execution::connect(sender | adaptor, receiver);
//  !o is true
std::execution::start(op);
//  o holds a variant whose active alternative is
//  std::exception_ptr
```

LSEG

Asynchronous Example

`set_error` allows asynchronous operations to transmit failures.

```
std::optional<std::variant<int, std::exception_ptr>> o;
const auto sender = std::execution::just(5);
const auto adaptor = std::execution::then([](const int i) {
  if (i > 10) {
    return i;
  }
  throw std::runtime_error("Less than or equal to 10");
});
const struct {
  using receiver_concept = std::execution::receiver_t;
  void set_value(const int i) && noexcept {
    o.emplace(i);
  }
  void set_error(std::exception_ptr ex) && noexcept {
    o.emplace(std::move(ex));
  }
  std::optional<std::variant<int, std::exception_ptr>>& o;
} receiver{o};
auto op = std::execution::connect(sender | adaptor, receiver);
//  !o is true
std::execution::start(op);
//  o holds a variant whose active alternative is
//  std::exception_ptr
```

LSEG

Asynchronous Example

Pipe syntax allows a sender (on the left) to be combined with a sender adaptor closure (on the right) to obtain a composed sender.

```
std::optional<std::variant<int, std::exception_ptr>> o;
const auto sender = std::execution::just(5);
const auto adaptor = std::execution::then([](const int i) {
  if (i > 10) {
    return i;
  }
  throw std::runtime_error("Less than or equal to 10");
});
const struct {
  using receiver_concept = std::execution::receiver_t;
  void set_value(const int i) && noexcept {
    o.emplace(i);
  }
  void set_error(std::exception_ptr ex) && noexcept {
    o.emplace(std::move(ex));
  }
  std::optional<std::variant<int, std::exception_ptr>>& o;
} receiver{o};
auto op = std::execution::connect(sender | adaptor, receiver);
//  !o is true
std::execution::start(op);
//  o holds a variant whose active alternative is
//  std::exception_ptr
```

LSEG

Question of Synchronous Equivalence

This is the equivalent of the asynchronous code on the preceding slide.

```
const auto f1 = []() noexcept { return 5; };
const auto f2 = [](auto f) {
  auto i = f();
  if (i > 10) {
    return i;
  }
  throw std::runtime_error("Less than or equal to 10");
};
//  Throws
(void)f2(f1);
```

Question of Synchronous Equivalence

This is not the synchronous equivalent of the asynchronous code that's been discussed.

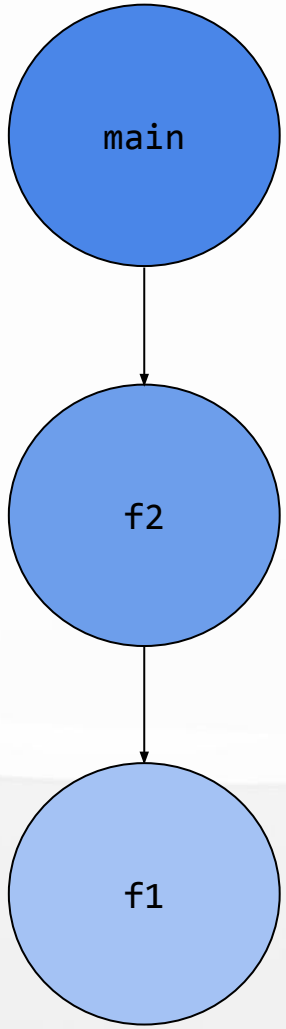
Why not?

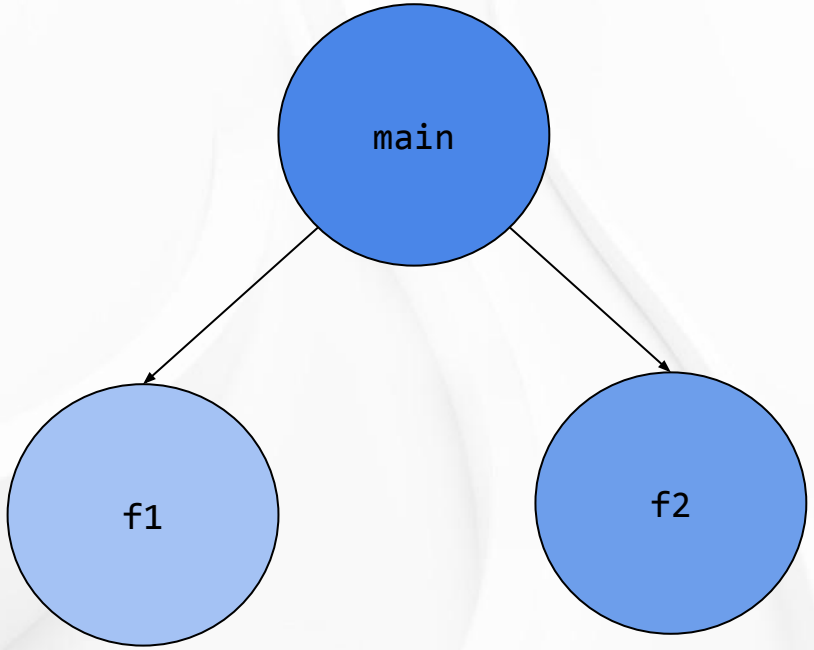
```
const auto f1 = []() noexcept { return 5; };
const auto f2 = [](auto f) {
  auto i = f();
  if (i > 10) {
    return i;
  }
  throw std::runtime_error("Less than or equal to 10");
};
//  Throws
(void)f2(f1);
```

```
const auto f1 = []() noexcept { return 5; };
const auto f2 = [](const int i) {
  if (i > 10) {
    return i;
  }
  throw std::runtime_error("Less than or equal to 10");
};
//  Throws
(void)f2(f1());
```

LSEG

Call Graphs



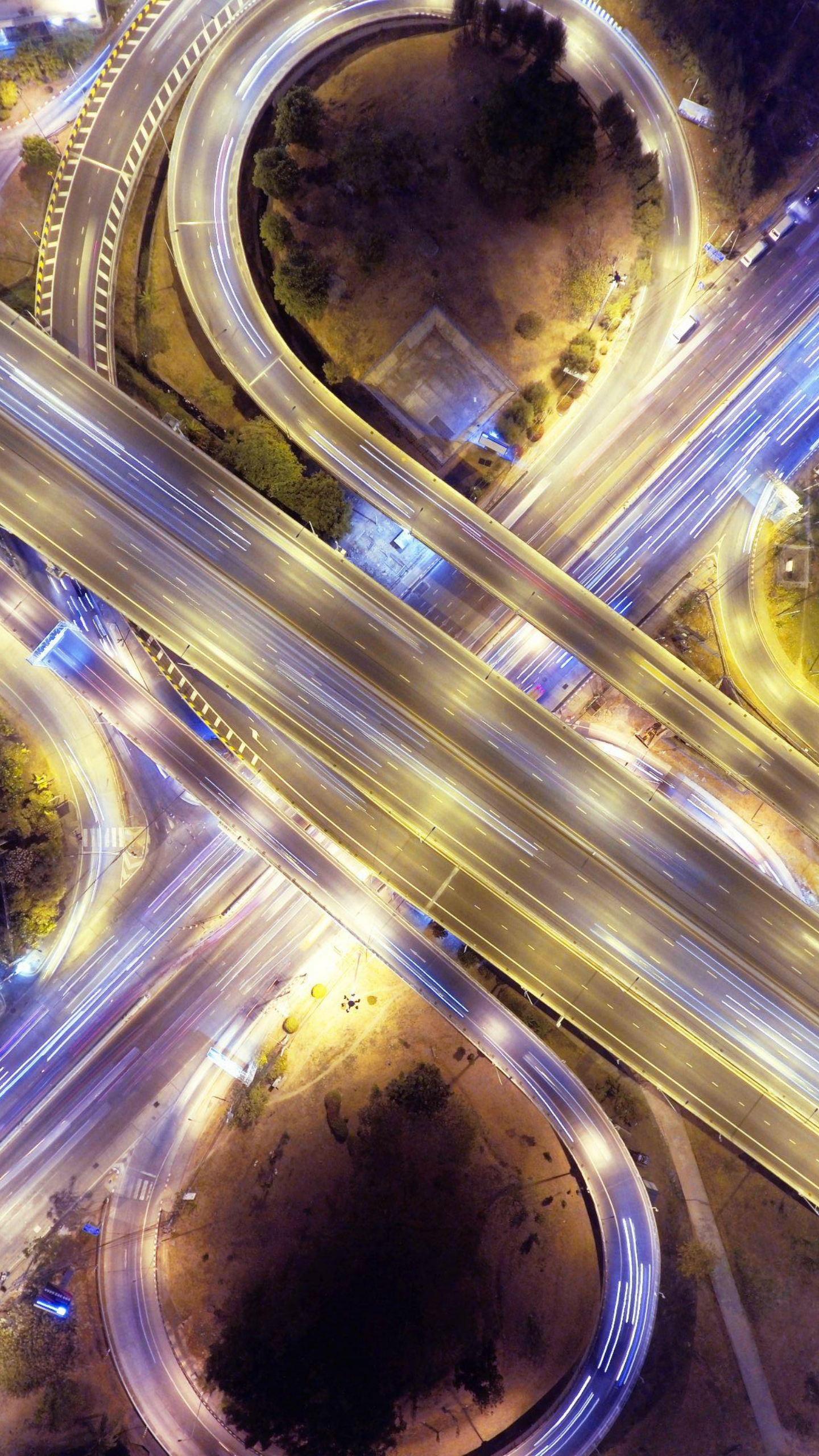


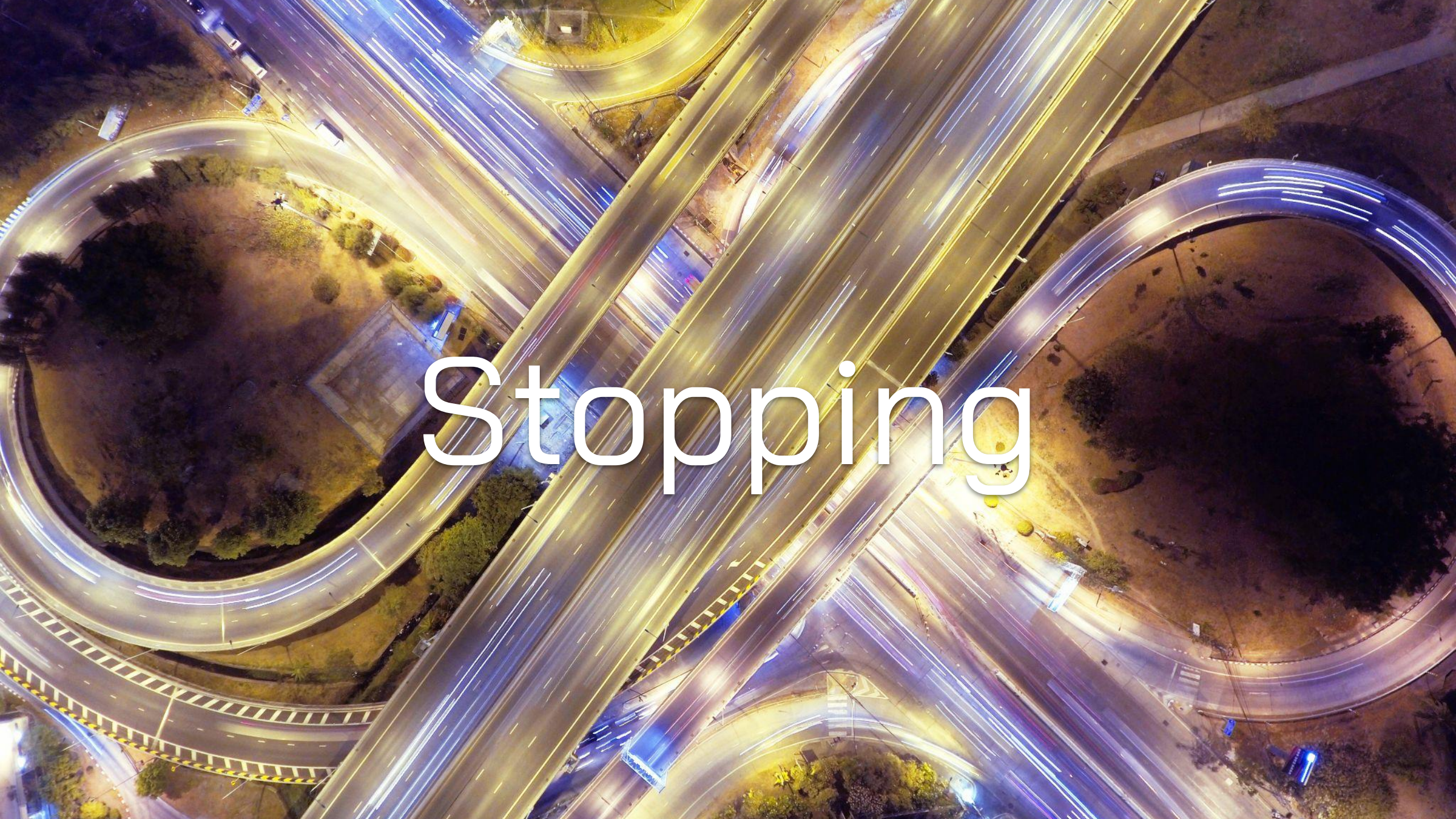

std::execution::let_value

Asynchronous Example

The use of `std::execution::let_value` here is contrived, but illustrates "calling" a "function" with the "return value" of a predecessor "function."

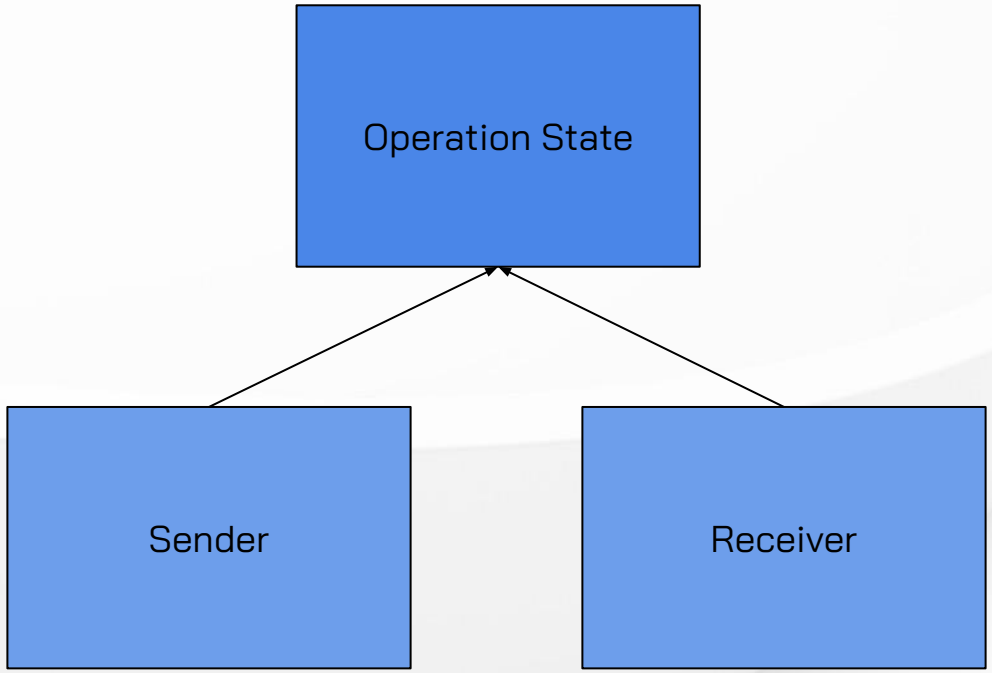
```
//  ...
auto op = std::execution::connect(
  std::execution::just(5) |
  std::execution::let_value([](const int& i) noexcept {
    return
      std::execution::just(i) |
      std::execution::then([](const int i) {
        if (i > 10) {
          return i;
        }
        throw std::runtime_error("Less than or equal to 10");
      });
  }),
  receiver);
//  ...
```



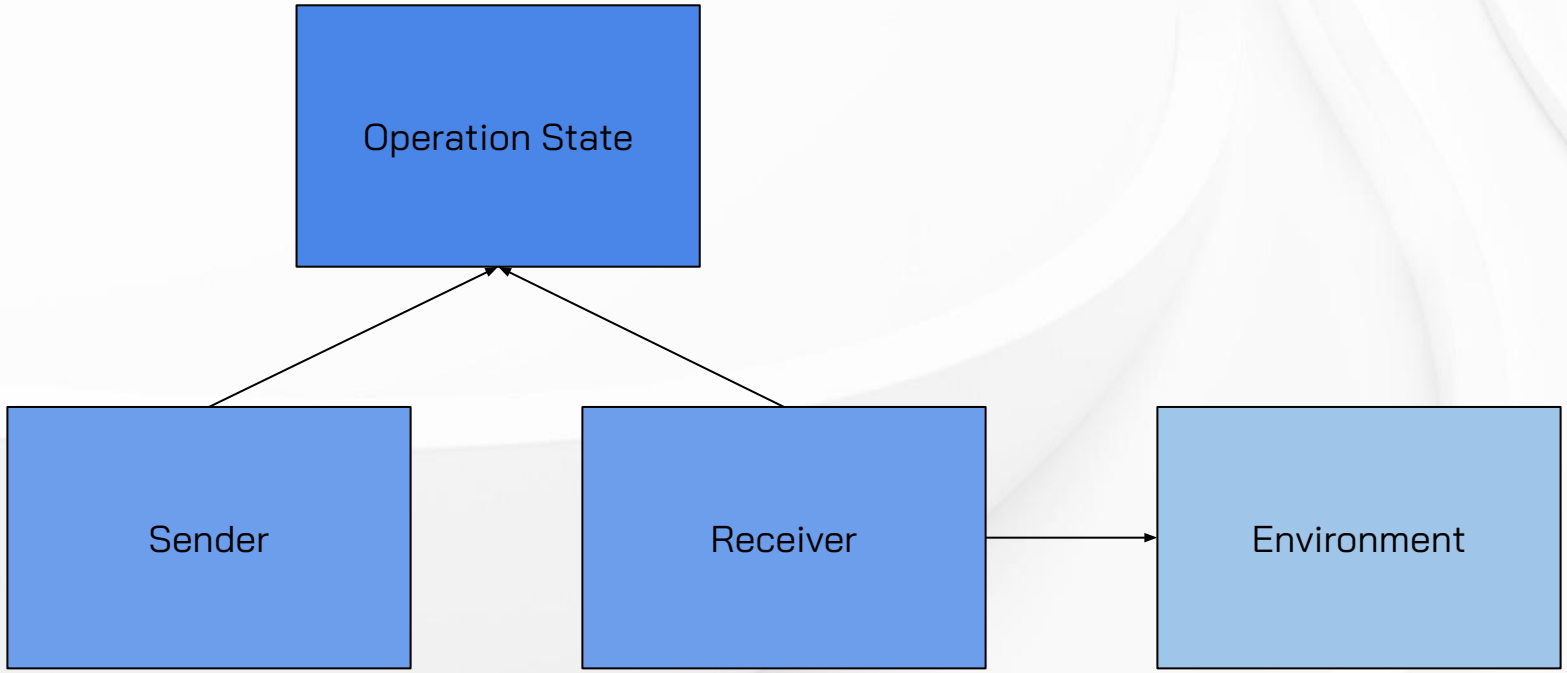

Stopping

How do I ask something to stop?

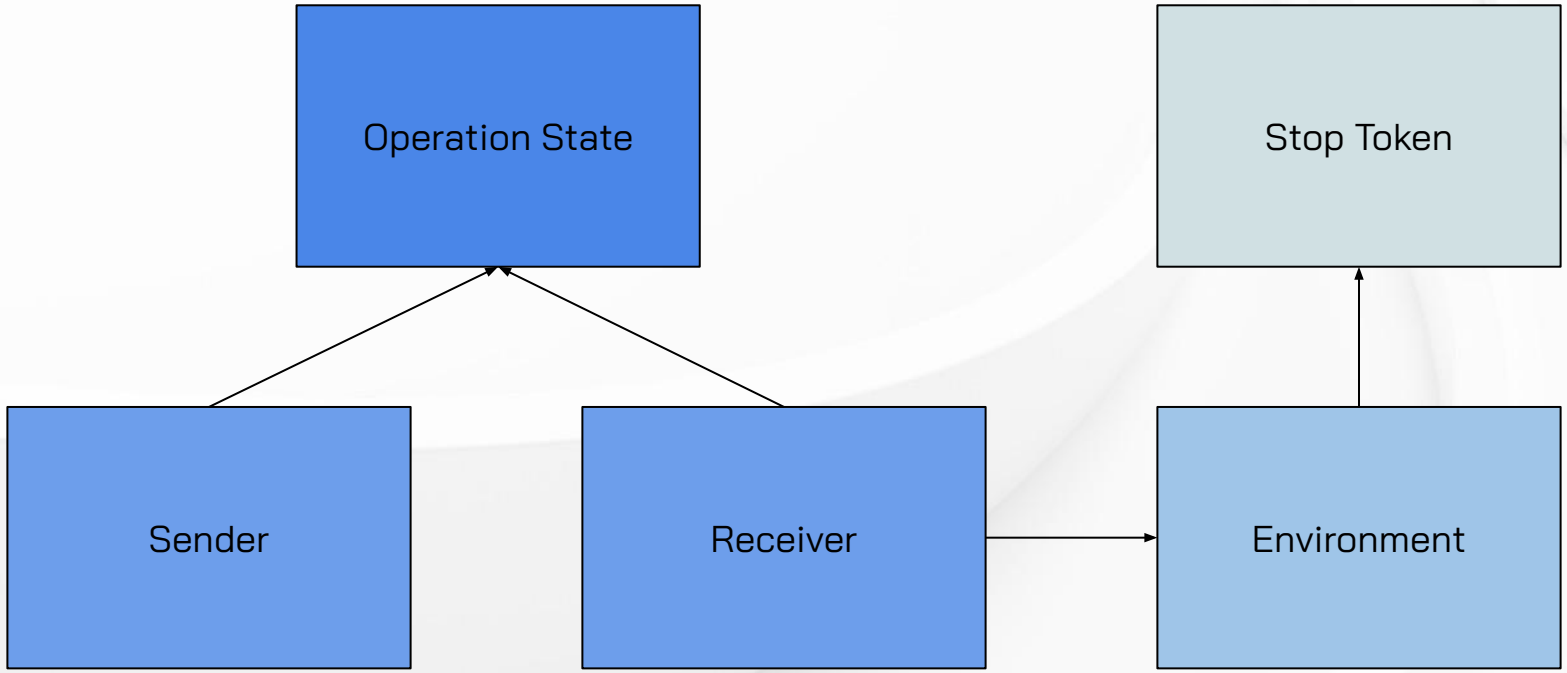
Sender & Receiver Ecosystem



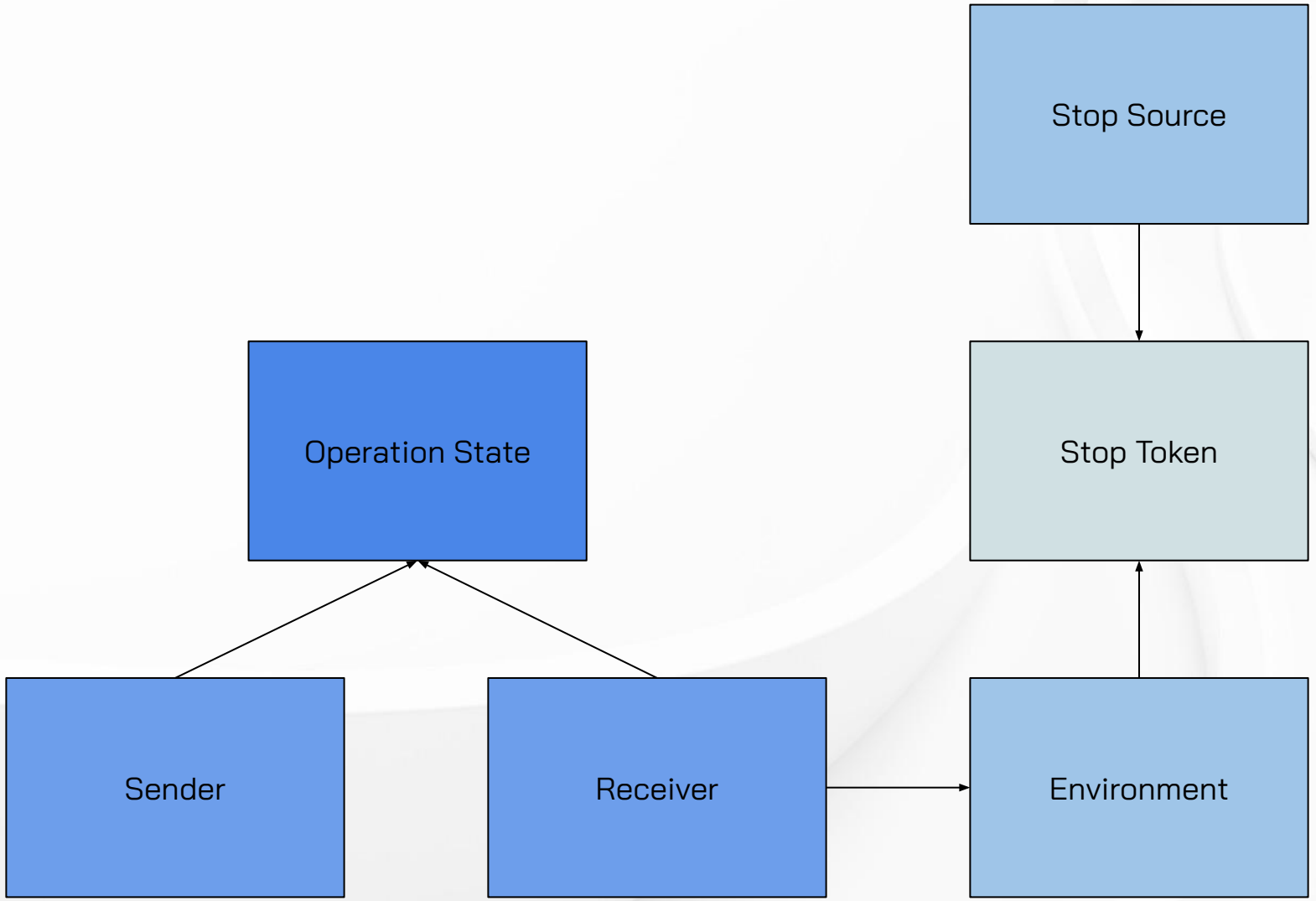
Sender & Receiver Ecosystem



Sender & Receiver Ecosystem



Sender & Receiver Ecosystem



std::stop_source

Defined in header <stop_token>

```
class stop_source;
```
(since C++20)

The stop_source class provides the means to issue a stop request, such as for std::jthread cancellation. A stop request made for one stop_source object is visible to all stop_sources and std::stop_tokens of the same associated stop-state; any std::stop_callback(s) registered for associated std::stop_token(s) will be invoked, and any std::condition_variable_any objects waiting on associated std::stop_token(s) will be awoken.

Once a stop is requested, it cannot be withdrawn. Additional stop requests have no effect.

Member functions

(constructor)	constructs new stop_source object (public member function)
(destructor)	destructs the stop_source object (public member function)
operator=	assigns the stop_source object (public member function)

Modifiers

request_stop	makes a stop request for the associated stop-state, if any (public member function)
swap	swaps two stop_source objects (public member function)

Observers

	returns a stop_token for the associated stop-state

What is a Stop Token?

From `std::stop_token` to `std::stoppable_token`

From P2300R10 (accepted into C++26 working draft in St. Louis):

An object of a type that models `stoppable_token` can be passed to an operation that can either

- *actively poll the token to check if there has been a stop request, or*
- *register a callback that will be called in the event that a stop request is made.*

Also from P2300R10:

```
template<class Token>
  concept stoppable_token =
    requires (const Token tok) {
      typename check-type-alias-exists<Token::template callback_type>;
      { tok.stop_requested() } noexcept -> same_as<bool>;
      { tok.stop_possible() } noexcept -> same_as<bool>;
      { Token(tok) } noexcept; // see implicit expression variations
                               // ([concepts.equality])
    } &&
    copyable<Token> &&
    equality_comparable<Token> &&
    swappable<Token>;
```

Sender/Receiver itself is the implementation of an `async-function`. An `async-function` can complete asynchronously with values, errors, or cancellation.

—P2849R0

Where’s the parallelism?

Parallelism is orthogonal to asynchrony

LSEG

Where’s the concurrency?

BYO

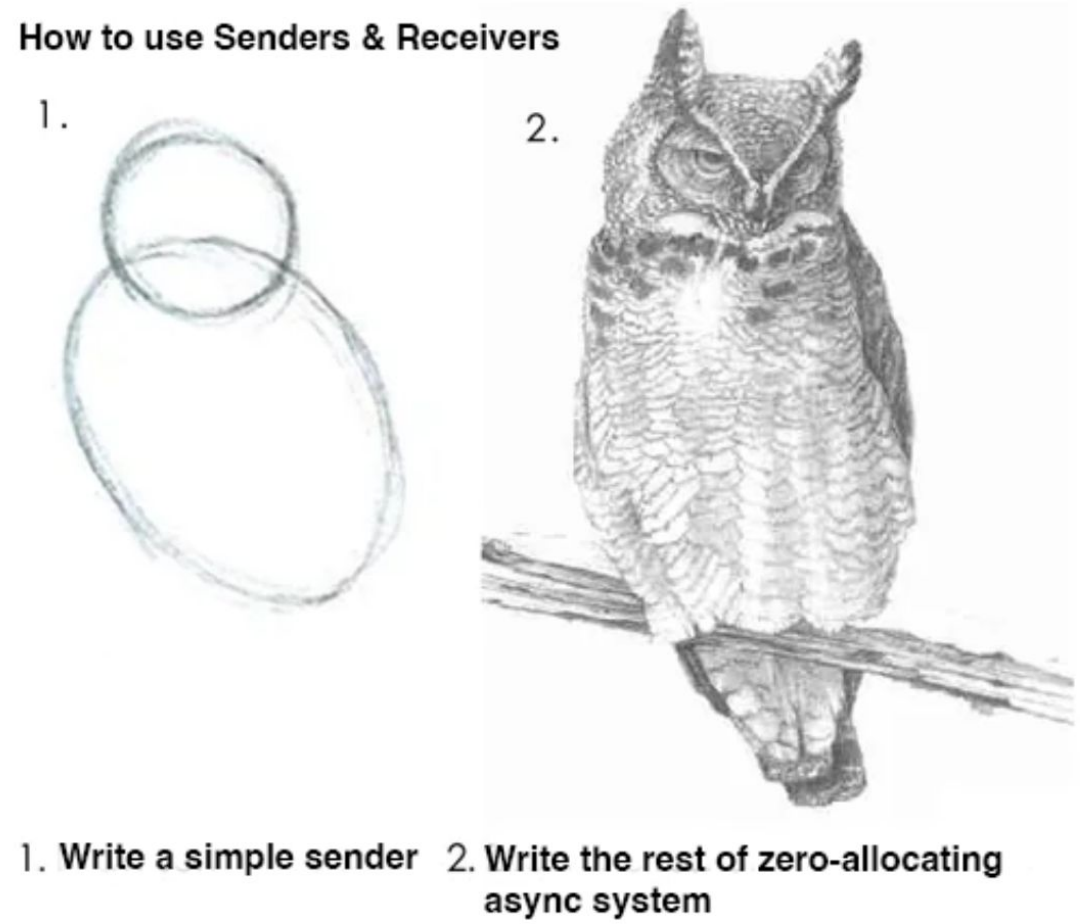
LSEG

This paper proposes a self-contained design for a Standard C++ framework for managing asynchronous execution on generic execution resources.

—P2300R10

Eric Niebler
@ericniebler

This remains my favorite tweet about senders.

> Hana Dusíková @hankadusikova · Feb 9
> I think I get it now.
>
> 
>

10:47 PM · Sep 26, 2024 · **2,928** Views

Deploying the Networking TS

ROBERT LEAHY

DataConn Message Structure

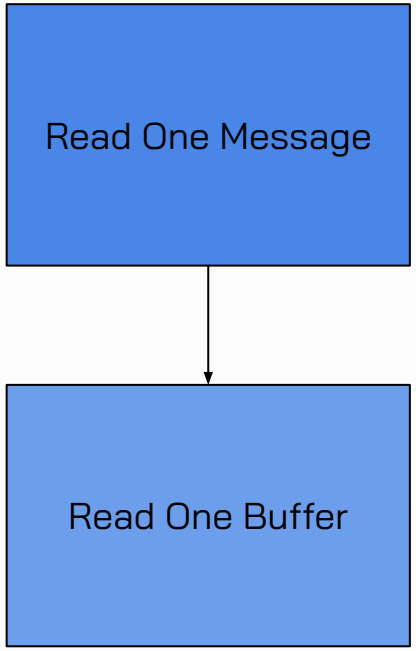
Offset	Field
0	Length (bytes) (little endian)
1	
2	
3	
4	Message type
...	Body (optional)

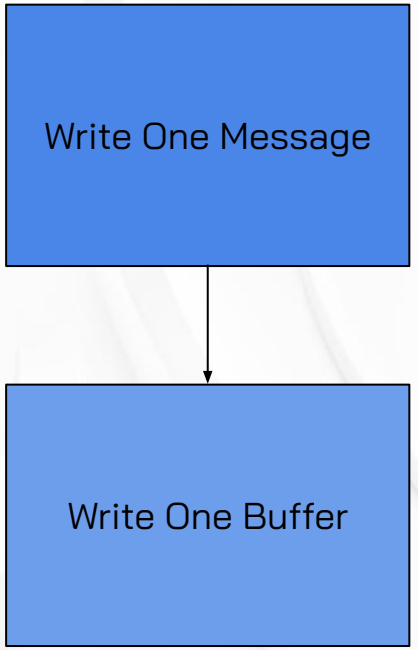
Where to Start?

Filling in the Blank Slate

Where to Start?

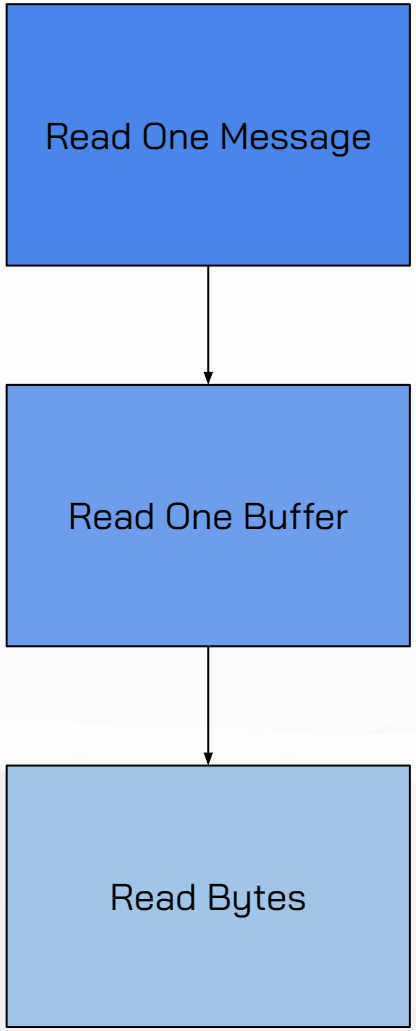
Filling in the Blank Slate

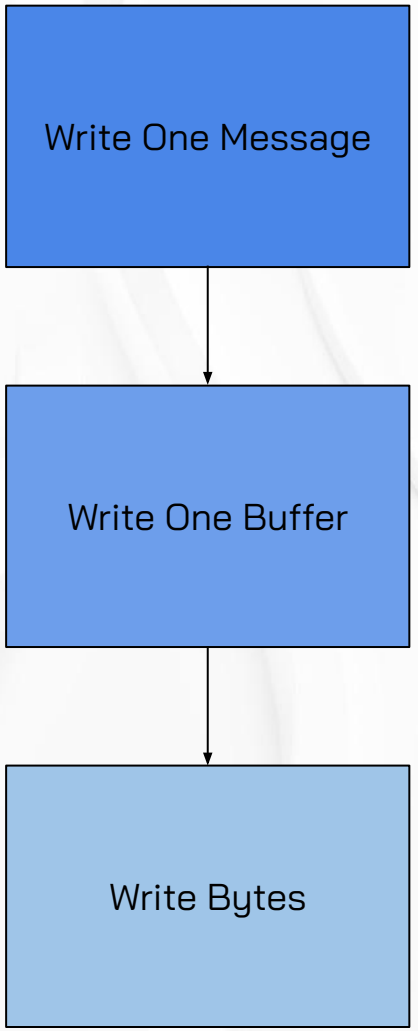




Where to Start?

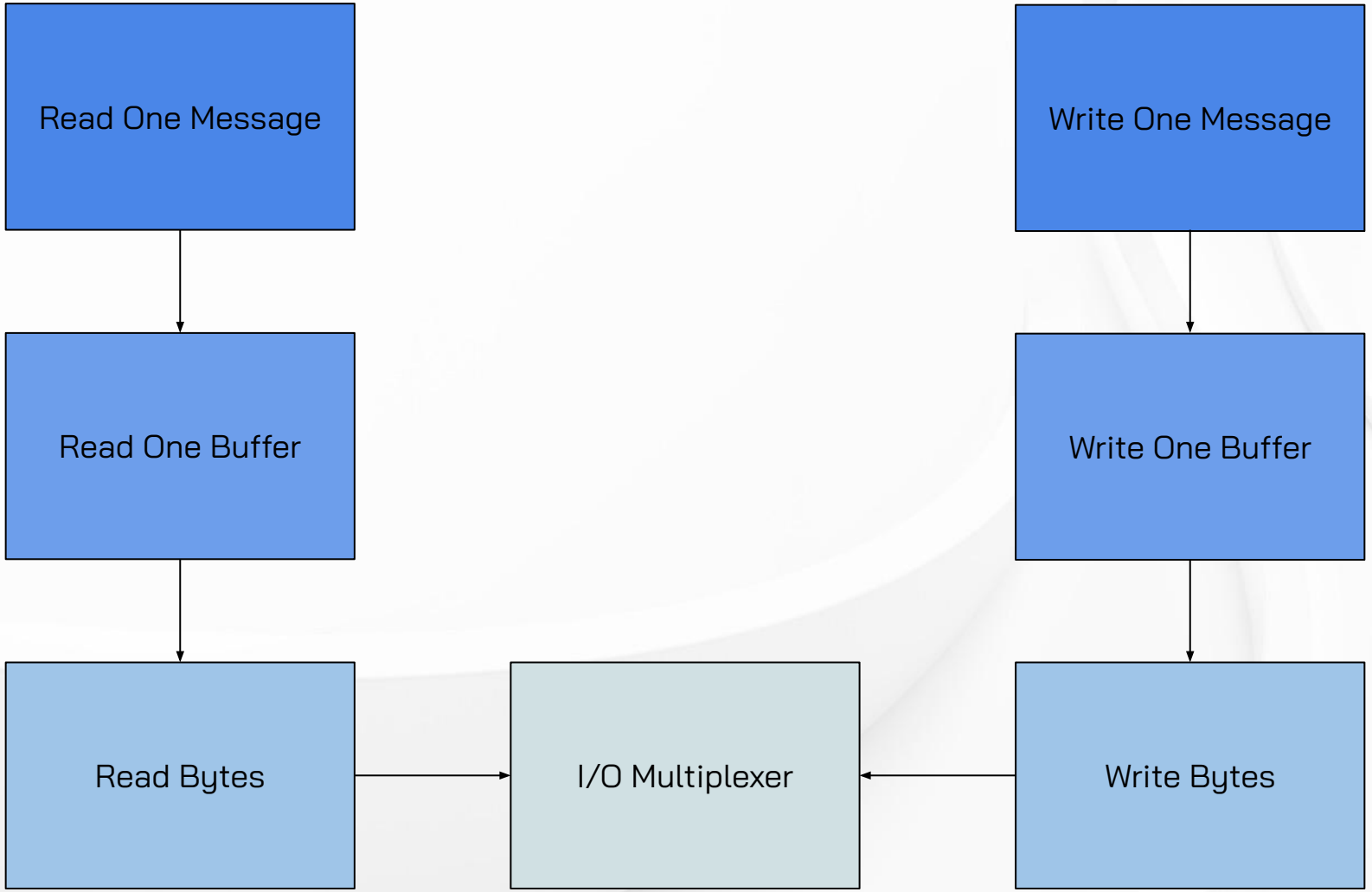
Filling in the Blank Slate





Where to Start?

Filling in the Blank Slate



NAME top

io_uring - Asynchronous I/O facility

SYNOPSIS top

```
#include <linux/io_uring.h>
```

DESCRIPTION top

io_uring is a Linux-specific API for asynchronous I/O. It allows the user to submit one or more I/O requests, which are processed asynchronously without blocking the calling process. **io_uring** gets its name from ring buffers which are shared between user space and kernel space. This arrangement allows for efficient I/O, while avoiding the overhead of copying buffers between them,

```
[rleahy@dev-nyc3-sv001 build_debug]$ uname -r
4.18.0-526.el8.x86_64
```

NAME top

epoll - I/O event notification facility

SYNOPSIS top

```
#include <sys/epoll.h>
```

DESCRIPTION top

The **epoll** API performs a similar task to poll(2): monitoring multiple file descriptors to see if I/O is possible on any of them. The **epoll** API can be used either as an edge-triggered or a level-triggered interface and scales well to large numbers of watched file descriptors.

Introduction to epoll

The second parameter is obsolescent but still needs to be non-zero.

```
int epoll_create(int size);
```

Introduction to epoll

Closure type.

We'll only be using the pointer so it's all that's shown here.

```
int epoll_create(int size);

union epoll_data {
  void* ptr;
  //  ...
};
```

Introduction to epoll

Used to both subscribe and transmit readiness.

```
int epoll_create(int size);

union epoll_data {
  void* ptr;
  //  ...
};

struct epoll_event {
  uint32_t          events;
  union epoll_data data;
};
```

Introduction to epoll

Updates interest.

op is `EPOLL_CTL_ADD`, `EPOLL_CTL_MOD`, or `EPOLL_CTL_DEL`.

```
int epoll_create(int size);

union epoll_data {
  void* ptr;
  //  ...
};

struct epoll_event {
  uint32_t          events;
  union epoll_data data;
};

int epoll_ctl(
  int epfd,
  int op,
  int fd,
  struct epoll_event* event);
```

Introduction to epoll

Blocks the current thread to dequeue some number of events.

```
int epoll_create(int size);

union epoll_data {
  void* ptr;
  //  ...
};

struct epoll_event {
  uint32_t          events;
  union epoll_data data;
};

int epoll_ctl(
  int epfd,
  int op,
  int fd,
  struct epoll_event* event);

int epoll_wait(
  int epfd,
  struct epoll_event* events,
  int maxevents,
  int timeout);
```

LSEG

Low Level Building Block

Minimum possible abstraction to adapt epoll into the `std::execution` ecosystem.

`wait` returns a sender which is a thin wrapper around `epoll_ctl`.

`run` is a thin wrapper around `epoll_wait`.

```
struct epoll {
  epoll();
  using native_handle_type = int;
  native_handle_type native_handle() const noexcept;
  using events_type = std::uint32_t;
  std::execution::sender auto wait(
    native_handle_type fd,
    events_type events) noexcept;
  void run() noexcept;
};
```

LSEG

The Easy Parts

`exec::safe_file_descriptor` simply provides an RAII wrapper around a POSIX file descriptor and is part of nVidia's "stdexec" library.

```
struct epoll {
  epoll()
    : fd_([&]() {
        auto retr = ::epoll_create(1);
        if (retr == -1) {
          throw system_error("epoll_create(1)");
        }
        return retr;
      }())
  {}
  native_handle_type native_handle() const noexcept {
    return fd_.native_handle();
  }
private:
  ::exec::safe_file_descriptor fd_;
  /*  ...
   *
   *
   *
   *
   *
   *
   *
   *
   */
};
```

LSEG

Error Reporting

`std::system_error` exists, but what about this unqualified `system_error`?

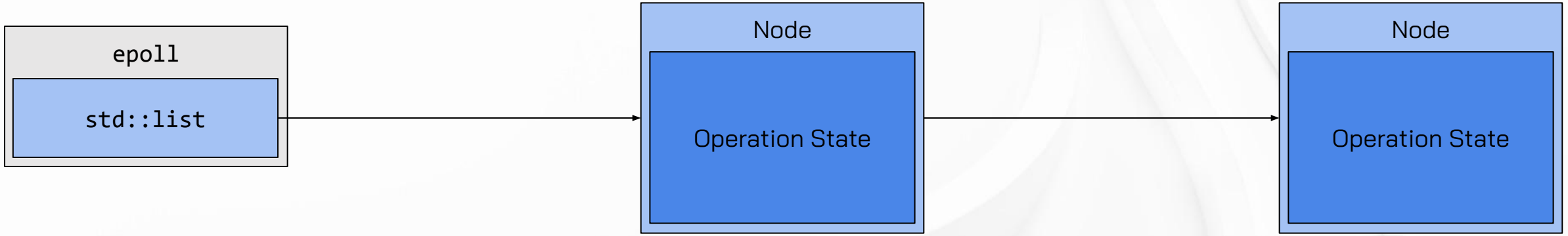
```
struct epoll {
  epoll()
    : fd_([&]() {
        auto retr = ::epoll_create(1);
        if (retr == -1) {
          throw system_error("epoll_create(1)");
        }
        return retr;
      }())
  {}
  native_handle_type native_handle() const noexcept {
    return fd_.native_handle();
  }
private:
  ::exec::safe_file_descriptor fd_;
  /*  ...
   *
   *
   *
   *
   *
   *
   *
   *
   */
};
```

LSEG

```
epoll_create(1) failed with Too many open files (24)
```

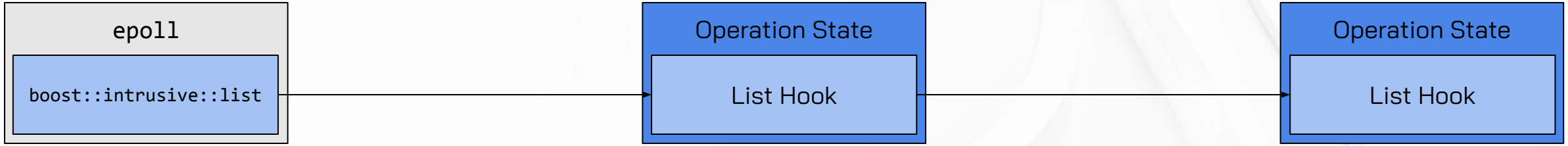
Linked List of Operations

Iteration is necessary to propagate errors



Intrusive Linked List of Operations

Avoiding allocation by leveraging operation states



Storing Operations

Intrusive linked list of operation states, or rather the common base class thereof.

```
struct epoll {
  //  ...
  struct waiting_tag_ {};
  using waiting_hook_ = ::boost::intrusive::list_base_hook<
    ::boost::intrusive::tag<waiting_tag_>>;
  struct base_ : waiting_hook_ { /* ... */ };
  using ops_type_ = ::boost::intrusive::list<
    base_,
    ::boost::intrusive::constant_time_size<false>,
    ::boost::intrusive::base_hook<waiting_hook_>>;
  ops_type_ ops_;
  /*  ...
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   */
};
```

LSEG

Type Erasure of Operations

`base_` provides a virtual interface to transmit error and value completions since the actual type of the operation state will be erased since it will depend on the receiver.

```
struct epoll {
  //  ...
  struct base_ : waiting_hook_ {
    explicit base_(const native_handle_type fd) noexcept
      : fd(fd)
    {}
    base_(const base_&) = delete;
    base_& operator=(const base_&) = delete;
    virtual void error(std::exception_ptr) noexcept = 0;
    virtual void value(events_type) noexcept = 0;
    native_handle_type fd;
  };
  /*  ...
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   */
};
```

LSEG

Operation State

Member variables.

```
struct epoll {
  //  ...
  template<typename Receiver>
  class operation_state_ : base_ {
    epoll& self_;
    events_type events_;
    Receiver receiver_;
    /*  ...
     *
     *
     *
     *
     *
     *
     *
     *
     *
     *
     *
     *
     *
     *
     */
  };
  //  ...
};
```

Operation State

Type alias for opt in concept.

```
struct epoll {
  //  ...
  template<typename Receiver>
  class operation_state_ : base_ {
    //  ...
  public:
    using operation_state_concept =
      std::execution::operation_state_t;
    /*  ...
     *
     *
     *
     *
     *
     *
     *
     *
     *
     *
     *
     *
     *
     */
  };
  //  ...
};
```

LSEG

Operation State

Constructor.

```
struct epoll {
  //  ...
  template<typename Receiver>
  class operation_state_ : base_ {
    //  ...
    explicit operation_state_(
      epoll& self,
      const native_handle_type fd,
      const events_type events,
      Receiver receiver) noexcept(
        std::is_nothrow_move_constructible_v<Receiver>)
      : base_(fd),
        self_(self),
        events_(events),
        receiver_(std::move(receiver))
    {}
    /*  ...
     *
     *
     *
     *
     */
  };
  //  ...
};
```

LSEG

Operation State

Type erasure of the value and error completion signals.

```
struct epoll {
  //  ...
  template<typename Receiver>
  class operation_state_ : base_ {
    //  ...
  private:
    void value(const events_type events) noexcept final {
      std::execution::set_value(
        std::move(receiver_),
        events));
    }
    void error(std::exception_ptr ex) noexcept final {
      std::execution::set_error(
        std::move(receiver_),
        std::move(ex));
    }
  public:
    /*  ...
     *
     *
     *
     */
  };
  //  ...
};
```

LSEG

Operation State

Since the sender & receiver model allows for one completion per operation `EPOLLONESHOT` is used to prevent repeat completions.

```
struct epoll {
  //  ...
  template<typename Receiver>
  class operation_state_ : base_ {
    //  ...
    void start() & noexcept {
      ::epoll_event e{};
      e.events = events_ | EPOLLONESHOT;
      e.data.ptr = this;
      /*  ...
       *
       *
       *
       *
       *
       *
       *
       *
       *
       *
       *
       */
    }
  };
  //  ...
};
```

LSEG

Operation State

Failure is passed down to the receiver, not transmitted as an exception from `start`.

```
struct epoll {
  //  ...
  template<typename Receiver>
  class operation_state_ : base_ {
    //  ...
    void start() & noexcept {
      //  ...
      const auto fail = [&](const char* const str) noexcept {
        try {
          throw system_error(str);
        } catch (...) {
          error(std::current_exception());
        }
      };
      /*  ...
       *
       *
       *
       *
       *
       */
    }
  };
  //  ...
};
```

LSEG

Operation State

Partially-bound version of `epoll_ctl`.

```
struct epoll {
  //  ...
  template<typename Receiver>
  class operation_state_ : base_ {
    //  ...
    void start() & noexcept {
      //  ...
      const auto ctl = [&](const int op) noexcept {
        return ::epoll_ctl(self_.native_handle(), op, fd, &e);
      };
      /*  ...
       *
       *
       *
       *
       *
       *
       *
       *
       *
       *
       */
    }
  };
  //  ...
};
```

Operation State

If `epoll_ctl` succeeds we add ourselves to the intrusive linked list and the operation is in progress.

If `epoll_ctl` fails that completes the operation in error immediately.

```
struct epoll {
  //  ...
  template<typename Receiver>
  class operation_state_ : base_ {
    //  ...
    void start() & noexcept {
      //  ...
      if (ctl(EPOLL_CTL_MOD) == -1) {
        if (errno != ENOENT) {
          fail("epoll_ctl(..., EPOLL_CTL_MOD, ...)");
          return;
        }
        if (ctl(EPOLL_CTL_ADD) == -1) {
          fail("epoll_ctl(..., EPOLL_CTL_ADD, ...)");
          return;
        }
      }
      self_.ops_.push_back(*this);
    }
  };
  /*  ...
   *
   *
   */
};
```

Sender

Sender encapsulates the arguments (its members) to the asynchronous function it can be used to call.

```
struct epoll {
  //  ...
  class sender_ {
    epoll& self_;
    native_handle_type fd_;
    events_type events_;
  public:
    explicit sender_(
      epoll& self,
      const native_handle_type fd,
      const events_type events) noexcept
      : self_(self),
        fd_(fd),
        events_(events)
    {}
    using sender_concept = std::execution::sender_t;
    /*  ...
     *
     *
     *
     *
     */
  };
  //  ...
};
```

LSEG

Sender

Senders also publish the completion signatures the associated operation generates.

```
struct epoll {
  //  ...
  class sender_ {
    //  ...
    using completion_signatures =
      std::execution::completion_signatures<
        std::execution::set_value_t(events_type),
        std::execution::set_error_t(std::exception_ptr)>;
    /*  ...
     *
     *
     *
     *
     *
     *
     *
     *
     *
     *
     *
     *
     */
  };
  //  ...
};
```

LSEG

Sender

Sender encapsulates the function it calls via the operation state type returned from `connect` when the return point (i.e. the receiver) is provided.

```
struct epoll {
  //  ...
  class sender_ {
    //  ...
    template<
      std::execution::receiver_of<
        completion_signatures> Receiver>
    auto connect(Receiver receiver) const noexcept(
      std::is_nothrow_constructible_v<
        operation_state_<Receiver>,
        epoll&,
        const native_handle_type&,
        const events_type&,
        Receiver>)
    {
      return operation_state_<Receiver>(
        self_,
        fd_,
        events_,
        std::move(receiver));
    }
  };
  /*  ...
   */
};
```

LSEG

Sender Factory

Simply curries arguments into a sender.

```
struct epoll {
  //  ...
public:
  sender_ wait(
    const native_handle_type fd,
    const events_type events) noexcept
  {
    return sender_(*this, fd, events);
  }
  /*  ...
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   */
};
```

LSEG

Forward Progress

Everything up until this slide simply sets work up to be done, it does nothing to make forward progress on that work, which is where run comes in.

Here we pull completions (i.e. readiness notifications) out of the epoll file descriptor.

```
struct epoll {
  //  ...
  void run() noexcept {
    try {
      while (!ops_.empty()) {
        constexpr std::size_t events = 64;
        ::epoll_event buffer[64];
        const auto size = ::epoll_wait(
          fd_.native_handle(), buffer, events, -1);
        if (size < 0) {
          if (errno == EINTR) {
            continue;
          }
          throw system_error("epoll_wait");
        }
        /*  ...
         *
         *
         */
      }
    } catch (...) {
      //  ...
    }
  }
};
```

LSEG

Forward Progress

For each event returned by the poll file descriptor we obtain the associated operation state and complete the associated operation.

```
struct epoll {
  //  ...
  void run() noexcept {
    try {
      while (!ops_.empty()) {
        //  ...
        using std::begin;
        std::for_each(
          begin(buffer),
          begin(buffer) + size,
          [&](const ::epoll_event& event) {
            auto&& op = *static_cast<base_*>(event.data.ptr);
            ops_.erase(ops_.iterator_to(op));
            op.value(event.events);
          });
        }
      }
    } catch (...) {
      /*  ...
       *
       *
       */
    }
  }
};
```

LSEG

Forward Progress

If an exception is thrown due to some system call failing we fail all operations with that error.

```
struct epoll {
  //  ...
  void run() noexcept {
    try {
      //  ...
    } catch (...) {
      while (!ops_.empty()) {
        auto&& op = ops_.front();
        ops_.pop_front();
        op.error(std::current_exception());
      }
    }
  }
};
```

LSEG


Waiting Indefinitely

NAME top

eventfd - create a file descriptor for event notification

LIBRARY top

Standard C library (*libc*, *-lc*)

SYNOPSIS top

```
#include <sys/eventfd.h>

int eventfd(unsigned int initval, int flags);
```

DESCRIPTION top

Helper

Abstraction for eventfd.

```
struct eventfd : ::exec::safe_file_descriptor {
  explicit eventfd() : ::exec::safe_file_descriptor([&]() {
    auto retr = ::eventfd(0, 0);
    if (retr == -1) {
      throw system_error("eventfd");
    }
    return retr;
  }()) {}
  //  ...
};
```

Helper

Writing increments the eventfd's internal counter.

```
struct eventfd : ::exec::safe_file_descriptor {
  //  ...
  void write(const std::uint64_t val) {
    if (::write(native_handle(), &val, sizeof(val)) == -1) {
      throw system_error("write(eventfd, ...)");
    }
  }
  /*  ...
   */
};
```

Helper

Reading decrements the eventfd's counter the amount read..

```
struct eventfd : ::exec::safe_file_descriptor {
  //  ...
  std::uint64_t read() {
    std::uint64_t retr;
    if (::read(native_handle(), &retr, sizeof(retr)) == -1) {
      throw system_error("read(eventfd, ...)");
    }
    return retr;
  }
};
```

LSEG

Stop Support

Adding an `eventfd` member to the `epoll` class isn't helpful unless it's wired into the epoll file descriptor.

Note `EPOLLONESHOT` is not used here: We always want to be notified when the eventfd is readable.

```
struct epoll {
  epoll()
    : fd_([&]() {
        //  ...
      }())
  {
    ::epoll_event e{};
    e.events = EPOLLIN;
    if (::epoll_ctl(
      fd_.native_handle(),
      EPOLL_CTL_ADD,
      wake_up_.native_handle(),
      &e) == -1)
    {
      throw system_error(
        "epoll_ctl(..., EPOLL_CTL_ADD, eventfd, ...)");
    }
  }
private:
  eventfd wake_up_;
  /*  ...
   *
   *
   */
};
```

LSEG

Stop Support

Another linked list will be maintained to record operations which have requested to be stopped.

Since stop requests can come from any thread there is a `std::mutex` to guard the aforementioned list.

```
struct epoll {
  //  ...
  struct stopping_tag_ {};
  using stopping_hook_ = ::boost::intrusive::list_base_hook<
    ::boost::intrusive::tag<stopping_tag_>>;
  //  ...
  using stopping_type_ = ::boost::intrusive::list<
    base_,
    ::boost::intrusive::constant_time_size<false>,
    ::boost::intrusive::base_hook<stopping_hook_>>;
  mutable std::mutex m_;
  stopping_type_ stopping_;
  /*  ...
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   */
};
```

LSEG

Stop Support

`base_`, from which all operation states derive, gains a new virtual function: `stopped`, which delivers the stopped signal to the type erased receiver.

```
struct epoll {
  //  ...
  struct base_ : stopping_hook_, waiting_hook_ {
    //  ...
    virtual void stopped() noexcept = 0;
    //  ...
  };
  /*  ...
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   */
};
```

LSEG

Stop Support

Action which will be performed when stop is requested.

```
struct epoll {
  //  ...
  template<typename Receiver>
  class operation_state_ : base_ {
    struct on_stop_request_ {
      explicit on_stop_request_(operation_state_& self) noexcept
        : self_(self)
      {}
      void operator()() && noexcept {
        const std::lock_guard g(self_.self_.m_);
        self_.self_.stopping_.push_back(self_);
        self_.self_.wake_up_.write(1);
      }
    private:
      operation_state_& self_;
    };
    /*  ...
     *
     *
     *
     *
     */
  };
  //  ...
};
```

LSEG

Stop Support

Computation of stop callback type, and a member to store it.

```
struct epoll {
  //  ...
  template<typename Receiver>
  class operation_state_ : base_ {
    //  ...
    using stop_callback_ = std::stop_callback_for_t<
      std::stop_token_of_t<
        std::execution::env_of_t<
          Receiver>>,
      on_stop_request_>;
    std::optional<stop_callback_> stop_;
    /*  ...
     *
     *
     *
     *
     *
     *
     *
     *
     *
     *
     */
  };
  //  ...
};
```

LSEG

Stop Support

`release_` is needed to ensure value and error completion signals can be sent safely (i.e. without a race).

The value and error channels release before sending the signal to avoid races.

```
struct epoll {
  //  ...
  template<typename Receiver>
  class operation_state_ : base_ {
    //  ...
    void release_() noexcept {
      stop_.reset();
      if (stopping_hook_::is_linked()) {
        const std::lock_guard g(self_.m_);
        self_.stopping_.erase(
          self_.stopping_.iterator_to(*this));
      }
    }
    void value(const events_type events) noexcept final {
      release_();
      //  ...
    }
    void error(std::exception_ptr ex) noexcept final {
      release_();
      //  ...
    }
    //  ...
  };
  //  ...
};
```

LSEG

Why Release?

1. Operation completes with value or error
2. Stop requested
3. Stop callback runs
4. epoll file descriptor awakens and dispatches stopped
5. `set_stopped` invoked on the receiver

Why Release?

1. Operation completes with value or error
2. Stop requested
3. Stop callback runs
4. epoll file descriptor awakens and dispatches stopped
5. `set_stopped` invoked on the receiver

Multiple completion signals sent to the receiver

Stop Support

`stopped` provides type erasure for the stopped completion signal of the receiver.

```
struct epoll {
  //  ...
  template<typename Receiver>
  class operation_state_ : base_ {
    //  ...
    void stopped() noexcept final {
      std::execution::set_stopped(std::move(receiver_));
    }
    /*  ...
     *
     *
     *
     *
     *
     *
     *
     *
     *
     *
     *
     *
     *
     */
  };
  //  ...
};
```

LSEG

Stop Support

`start` needs to be extended to construct the stop callback after otherwise setting up the operation.

```
struct epoll {
  //  ...
  template<typename Receiver>
  class operation_state_ : base_ {
    //  ...
  public:
    void start() & noexcept {
      //  ...
      stop_.emplace(
        std::get_stop_token(
          std::execution::get_env(
            receiver_)),
        on_stop_request_(*this));
    }
  };
  /*  ...
   *
   *
   *
   *
   *
   *
   *
   *
   */
};
```

LSEG

Why Emplace?

1. Operation state and stop callback constructed
2. Stop requested
3. Stop callback runs
4. epoll file descriptor awakens and dispatches stopped
5. `set_stopped` invoked on the receiver

Why Emplace?

1. Operation state and stop callback constructed
2. Stop requested
3. Stop callback runs
4. epoll file descriptor awakens and dispatches stopped
5. `set_stopped` invoked on the receiver

The operation was never started

Stop Support

`std::execution::set_stopped_t()` signature is new.

```
struct epoll {
  //  ...
  class sender_ {
    //  ...
    using completion_signatures =
      std::execution::completion_signatures<
        std::execution::set_value_t(events_type),
        std::execution::set_error_t(std::exception_ptr)
        std::execution::set_stopped_t()>;
    /*  ...
     *
     *
     *
     *
     *
     *
     *
     *
     *
     *
     *
     *
     */
  };
  //  ...
};
```

LSEG

Stop Support

Null closure pointer indicates eventfd.

Defer stopped handling because stopped file descriptor may be later in buffer of dequeued events.

```
void run() noexcept {
  try {
    while (!ops_.empty()) {
      //  ...
      bool process = false;
      std::for_each(
        begin(buffer),
        begin(buffer) + size,
        [&](const ::epoll_event& event) {
          if (!event.data.ptr) {
            process = true;
            (void)wake_up_.read();
            return;
          }
          //  ...
        });
        /*  ...
         *
         *
         *
         *
         *
         *
         *
         *
         */
```

LSEG

Stop Support

Swapping collection reduces time lock is held.

```
    if (process) {
      stopping_type_ stopping;
      {
        const std::lock_guard g(m_);
        using std::swap;
        swap(stopping, stopping_);
      }
    }
  }
  //  ...
```

LSEG

Stop Support

Drain all stopping operations.

```
  if (process) {
    stopping_type_ stopping;
    {
      const std::lock_guard g(m_);
      using std::swap;
      swap(stopping, stopping_);
    }
    while (!stopping.empty()) {
      auto&& op = stopping.front();
      stopping.pop_front();
```

```
    }
  }
  //  ...
```

LSEG

Stop Support

Deregister so multiple completion signals can't be sent.

```
    if (process) {
      stopping_type_ stopping;
      {
        const std::lock_guard g(m_);
        using std::swap;
        swap(stopping, stopping_);
      }
      while (!stopping.empty()) {
        auto&& op = stopping.front();
        stopping.pop_front();
        if (::epoll_ctl(
          native_handle(),
          EPOLL_CTL_DEL,
          op.fd,
          nullptr) == -1)
        {
          throw system_error(
            "epoll_ctl(..., EPOLL_CTL_DEL, ...)");
        }


      }
    }
  }
  //  ...
```

LSEG

Stop Support

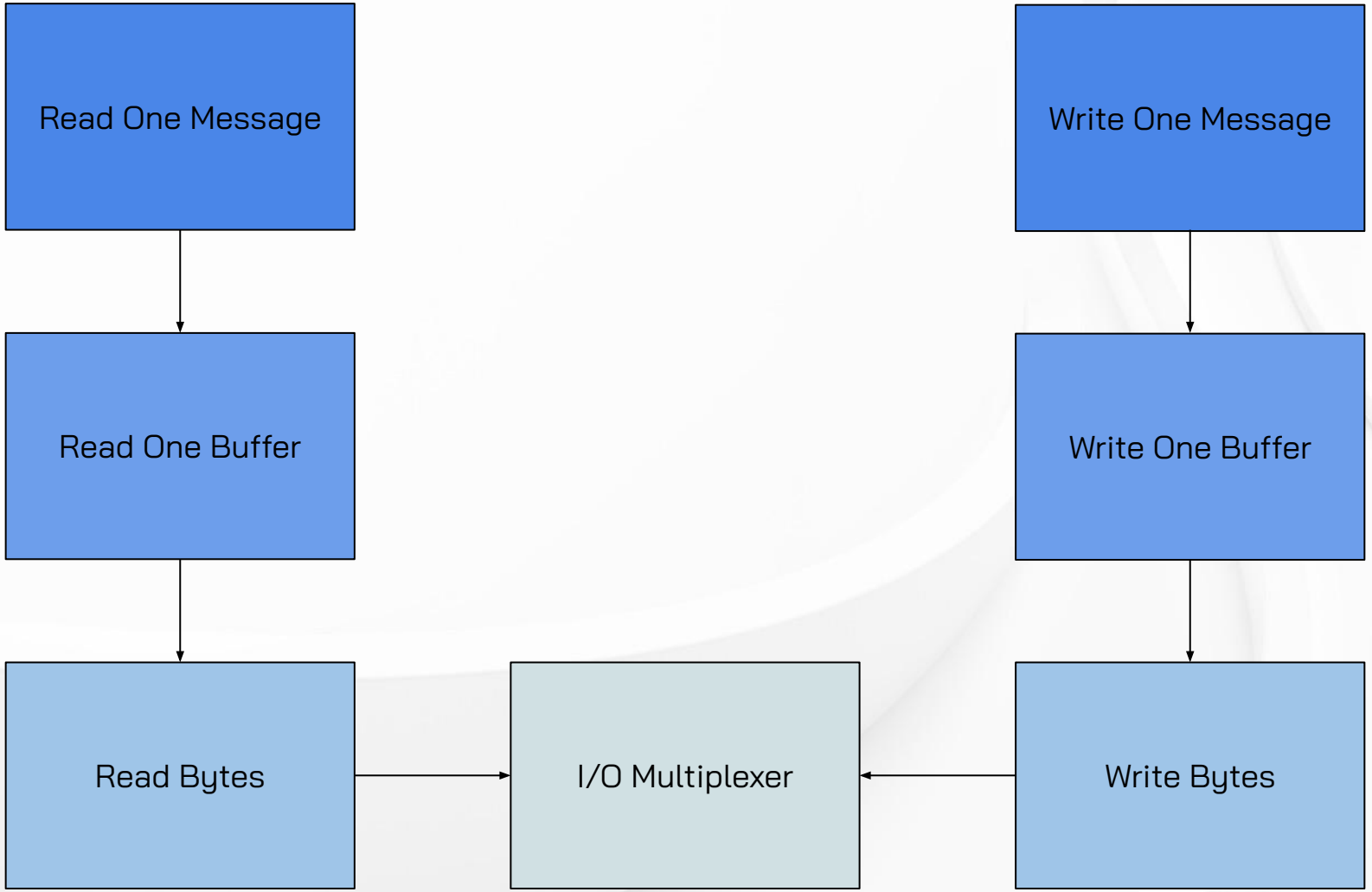
Send the completion signal.

```
    if (process) {
      stopping_type_ stopping;
      {
        const std::lock_guard g(m_);
        using std::swap;
        swap(stopping, stopping_);
      }
      while (!stopping.empty()) {
        auto&& op = stopping.front();
        stopping.pop_front();
        if (::epoll_ctl(
          native_handle(),
          EPOLL_CTL_DEL,
          op.fd,
          nullptr) == -1)
        {
          throw system_error(
            "epoll_ctl(..., EPOLL_CTL_DEL, ...)");
        }
        ops_.erase(ops_.iterator_to(op));
        op.stopped();
      }
    }
  }
  //  ...
```

LSEG

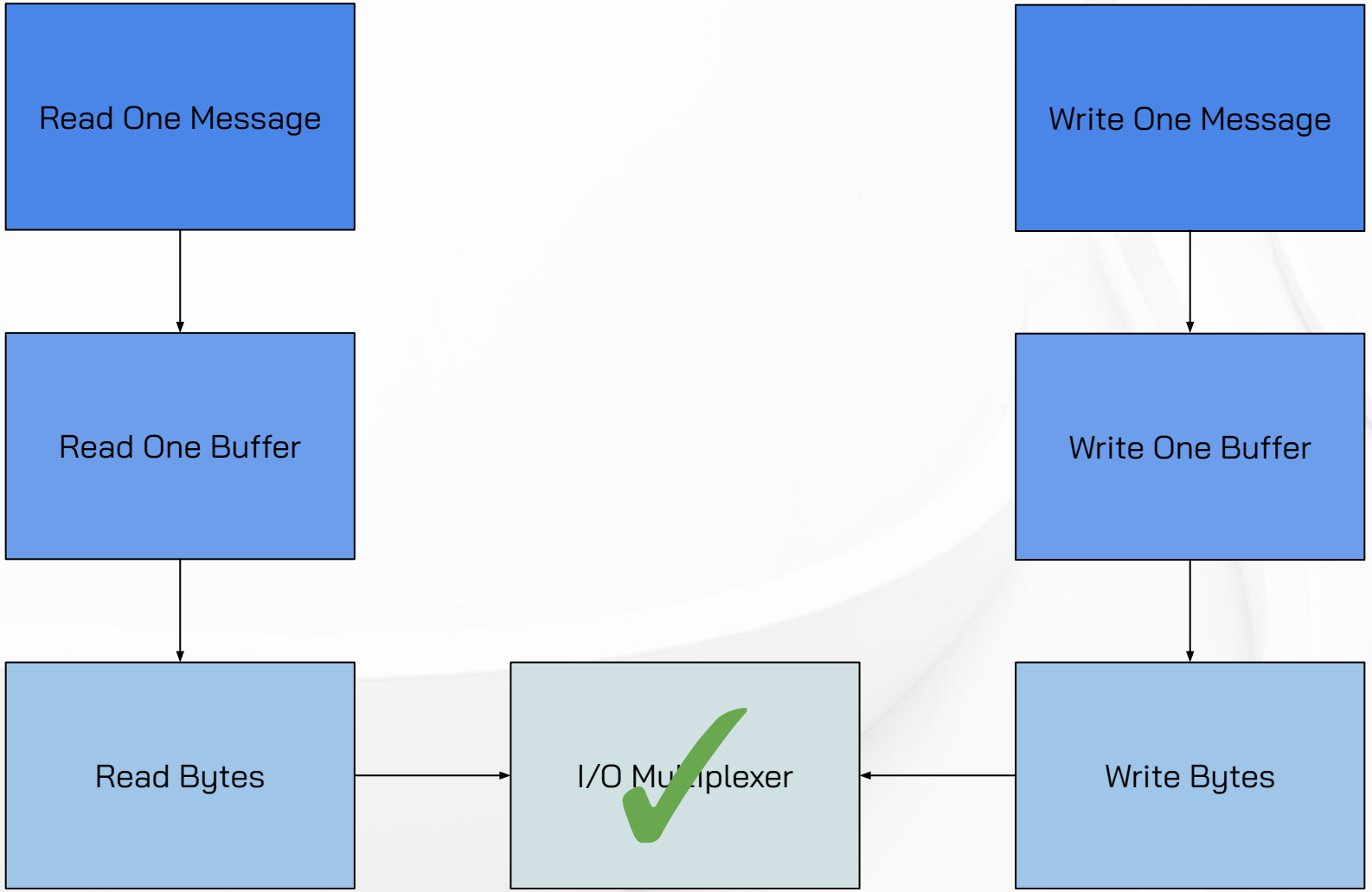
Where to Start?

Filling in the Blank Slate



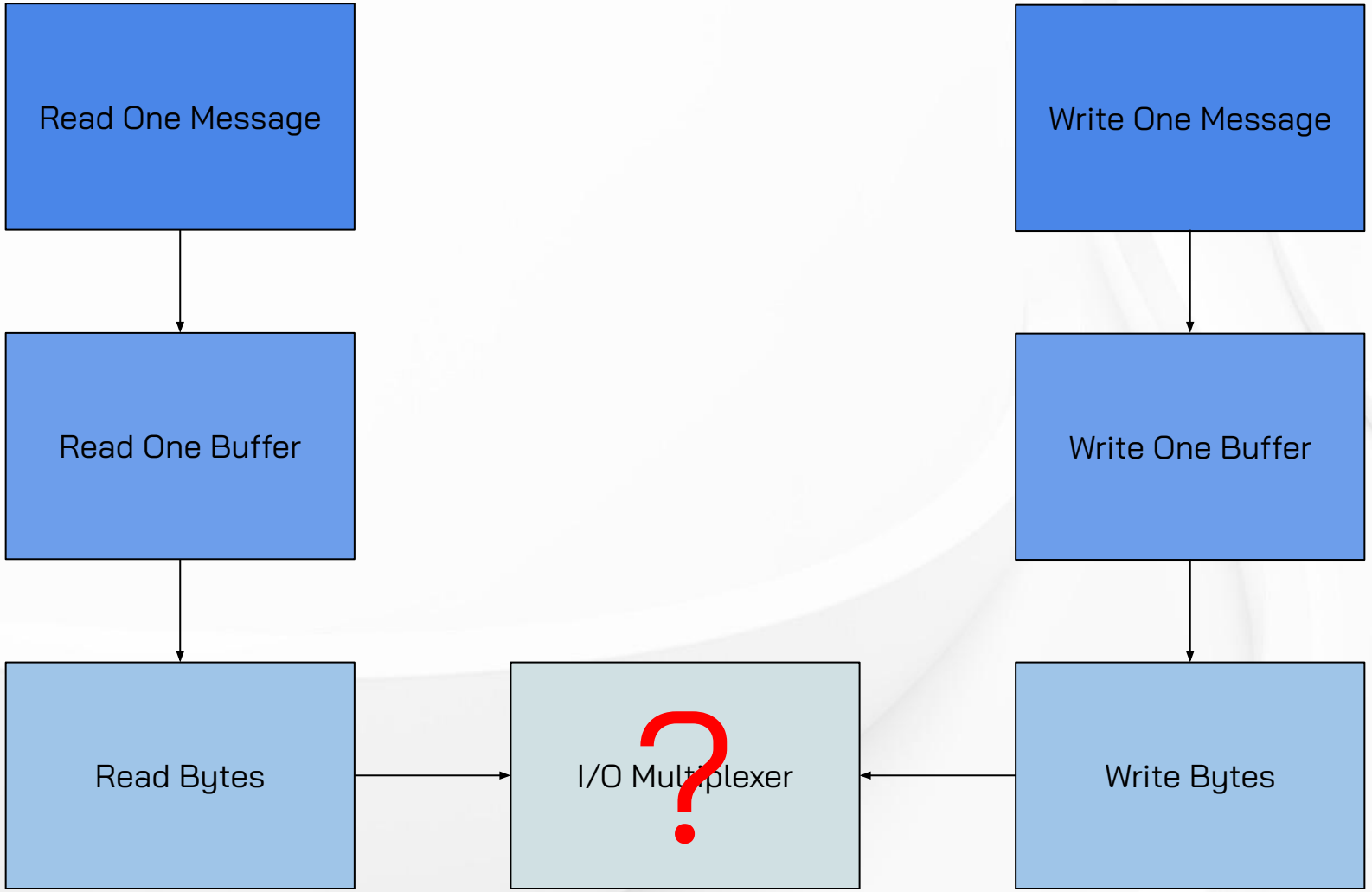
Where to Start?

Filling in the Blank Slate



Where to Start?

Filling in the Blank Slate



File Descriptor

Despite the fact epoll marries the two reading and writing are separate operations and `file_descriptor` models them as such.

```
struct file_descriptor : ::exec::safe_file_descriptor {
  explicit file_descriptor(epoll& ep, int fd) noexcept;
  explicit file_descriptor(
    epoll& ep,
    ::exec::safe_file_descriptor&& fd) noexcept;
  std::execution::sender auto wait_readable() noexcept;
  std::execution::sender auto wait_writable() noexcept;
};
```

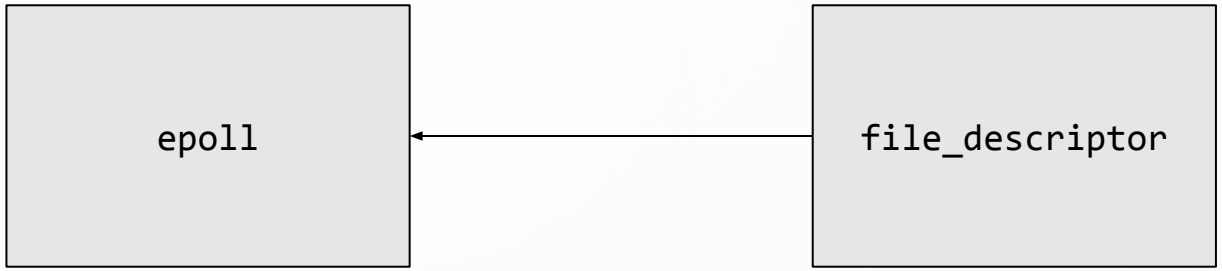
LSEG

LSEG


Core C++ 2024
Evolving C++ Networking With Senders & Receivers
Part 2
Robert Leahy

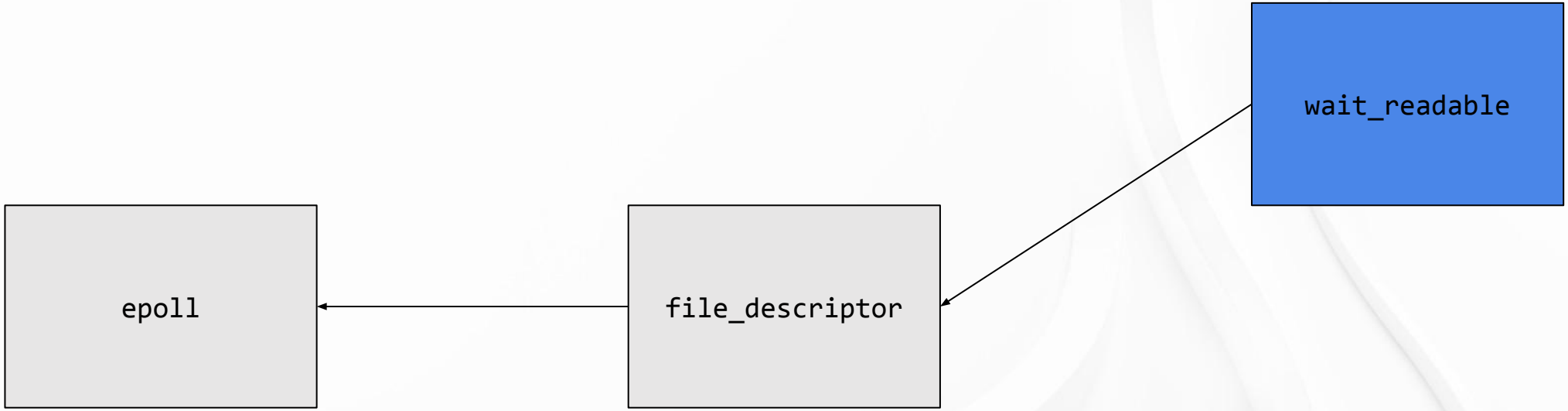
epoll & file_descriptor

A problem of demultiplexing



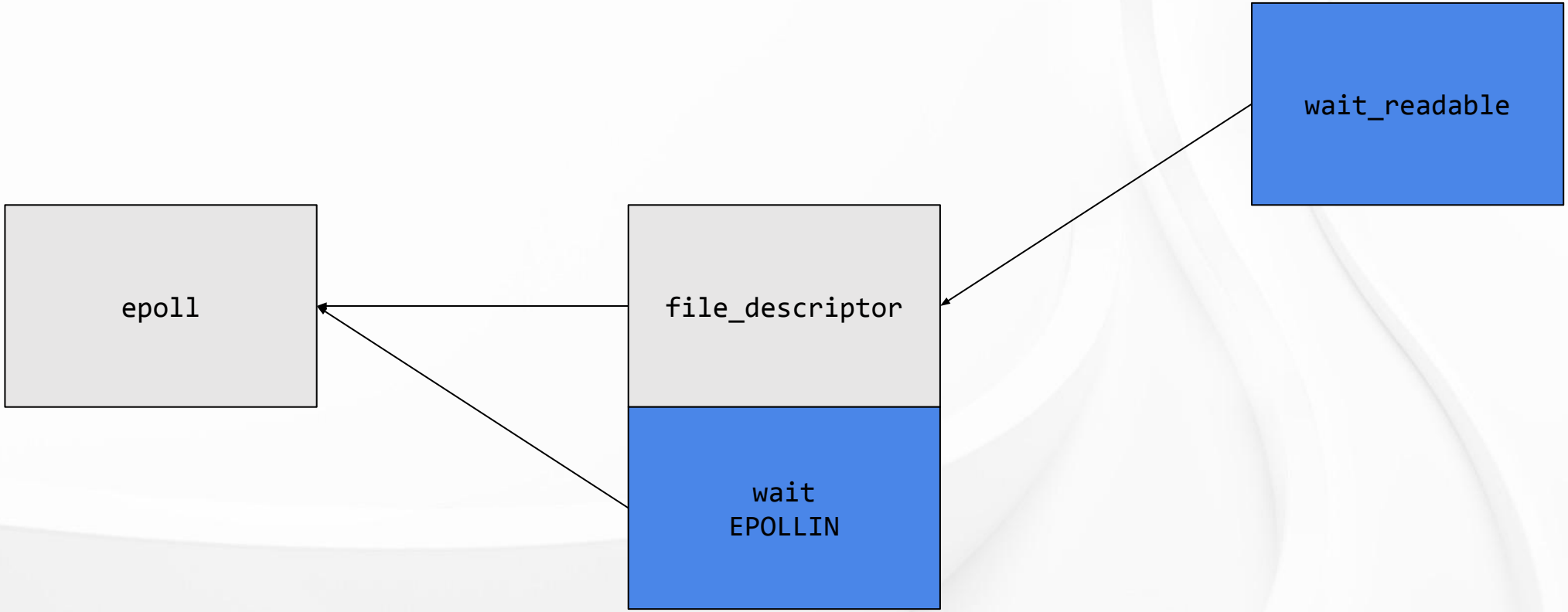
epoll & file_descriptor

A problem of demultiplexing



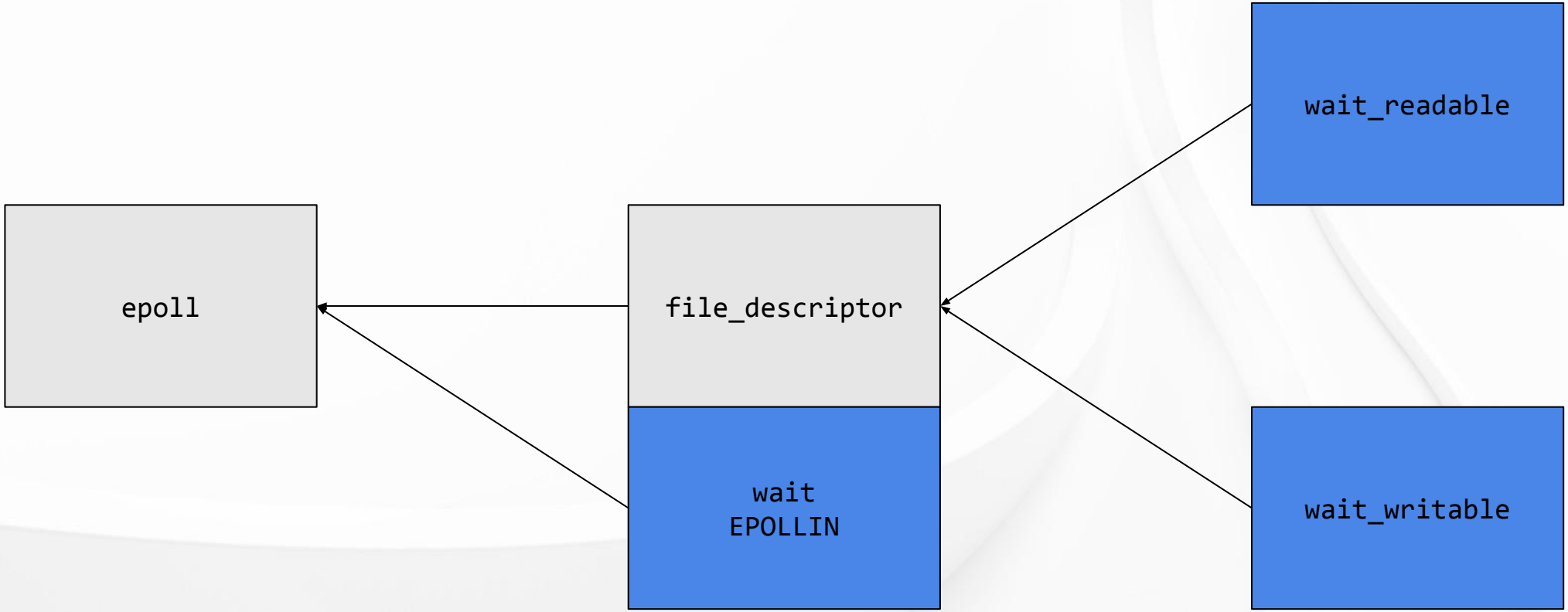
epoll & file_descriptor

A problem of demultiplexing



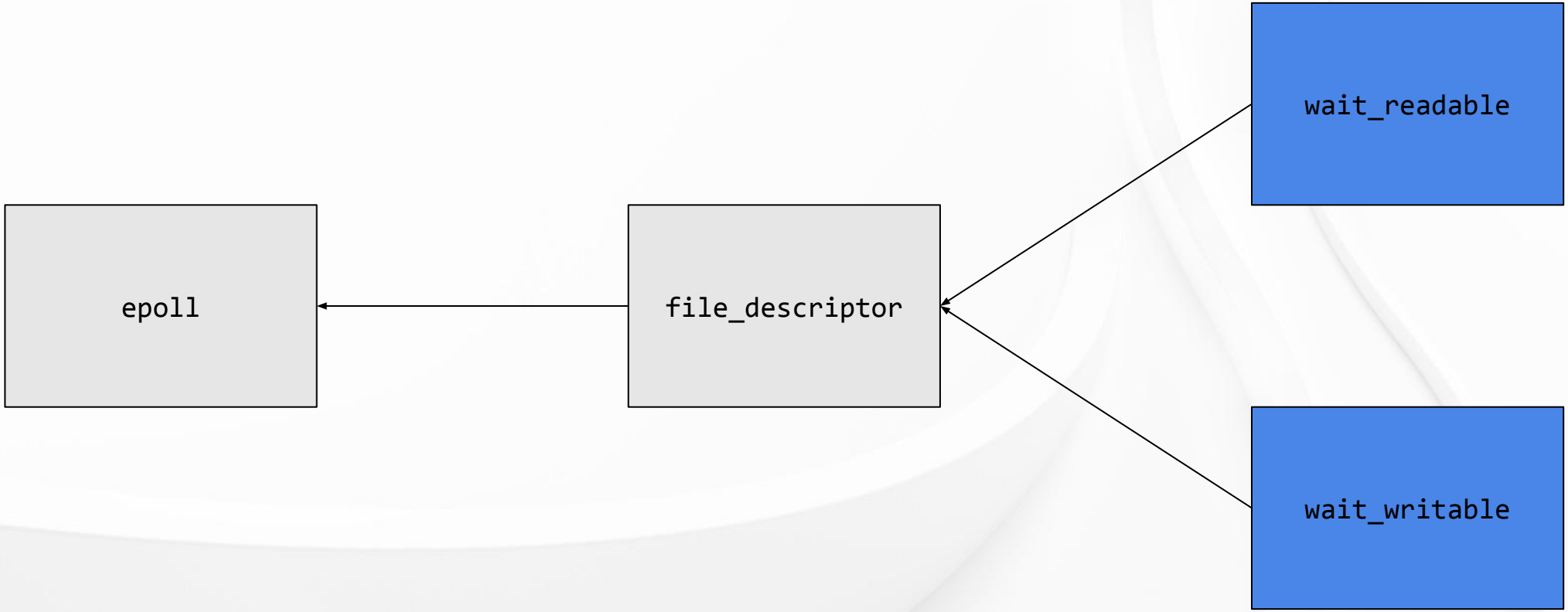
epoll & file_descriptor

A problem of demultiplexing



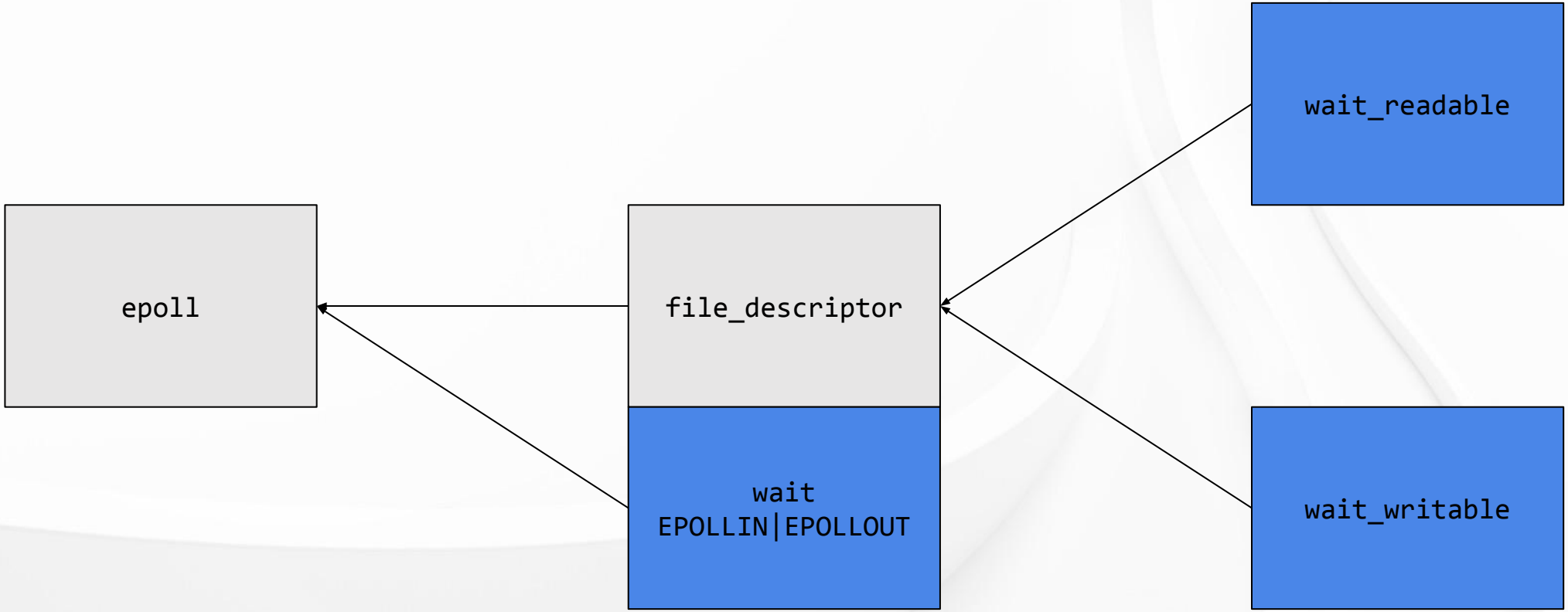
epoll & file_descriptor

A problem of demultiplexing



epoll & file_descriptor

A problem of demultiplexing



File Descriptor

`op_` is the currently in progress operation against the associated `epoll` (if any), which demultiplexes to `read_` and `write_`.

```
class file_descriptor : public ::exec::safe_file_descriptor {
  struct base_;
  struct receiver_ { /* ... */ };
  using inner_operation_state_ = std::execution::connect_result_t<
    decltype(std::declval<epoll&>().wait(0, 0)),
    receiver_>;
  epoll& ep_;
  std::optional<std::inplace_stop_source> stop_;
  base_* read_{nullptr};
  base_* write_{nullptr};
  std::optional<inner_operation_state_> op_;
  /*   ...
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   */
};
```

LSEG

File Descriptor

Operations waiting for readability and writability will type erase their receivers by deriving from `base_`.

```
class file_descriptor : public ::exec::safe_file_descriptor {
  //  ...
  struct base_ {
    base_() = default;
    base_(const base_&) = delete;
    base_& operator=(const base_&) = delete;
    virtual void value() noexcept = 0;
    virtual void error(std::exception_ptr) noexcept = 0;
    virtual void stop() noexcept = 0;
    virtual bool stop_requested() const noexcept = 0;
    virtual void acquire() noexcept = 0;
    virtual void release() noexcept = 0;
  };
  /*  ...
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   */
};
```

LSEG

File Descriptor

Boilerplate.

```
class file_descriptor : public ::exec::safe_file_descriptor {
  //  ...
public:
  explicit file_descriptor(
    epoll& ep,
    ::exec::safe_file_descriptor&& fd) noexcept
    : ::exec::safe_file_descriptor(std::move(fd)),
      ep_(ep)
  {}
  explicit file_descriptor(epoll& ep, const int fd) noexcept
    : file_descriptor(ep, ::exec::safe_file_descriptor(fd))
  {}
  /*  ...
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   */
};
```

LSEG

File Descriptor

When we begin handling a completion we reset our internal state by destroying the operation state for the now-ended operation, releasing the reference our sub operations have to the stop source, and then destroying the stop source (we must recreate it since stop sources can't be reset).

```
class file_descriptor : public ::exec::safe_file_descriptor {
  //  ...
  void begin_completion_() noexcept {
    op_.reset();
    if (read_) {
      read_->release();
    }
    if (write_) {
      write_->release();
    }
    stop_.reset();
  }
  /*  ...
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   */
};
```

LSEG

File Descriptor

Setting up stop sources is the first part of starting the parent operation.

```
class file_descriptor : public ::exec::safe_file_descriptor {
  //  ...
  void start_() noexcept {
    stop_.emplace();
    if (read_)
      read_->acquire();
    if (write_)
      write_->acquire();









  }
  /*  ...
   */
};
```

LSEG

File Descriptor

The needed flags are synthesized.

```
class file_descriptor : public ::exec::safe_file_descriptor {
  //  ...
  void start_() noexcept {
    stop_.emplace();
    if (read_)
      read_->acquire();
    if (write_)
      write_->acquire();
    epoll::events_type events = 0;
    if (read_)
      events |= EPOLLIN;
    if (write_)
      events |= EPOLLOUT;









  }
  /*  ...
   */
};
```

LSEG

File Descriptor

Operation state is created and started.

But how are we going to emplace an instance of an immovable type returned by a function?

```
class file_descriptor : public ::exec::safe_file_descriptor {
  //  ...
  void start_() noexcept {
    stop_.emplace();
    if (read_)
      read_->acquire();
    if (write_)
      write_->acquire();
    epoll::events_type events = 0;
    if (read_)
      events |= EPOLLIN;
    if (write_)
      events |= EPOLLOUT;
    op_.emplace(









          );
    std::execution::start(*op_);
  }
  /*  ...
   */
};
```

LSEG

```cpp
template<typename Function>
struct elide {
  explicit constexpr elide(Function f) noexcept(
    std::is_nothrow_move_constructible_v<Function>)
    : f_(std::move(f))
  {}
  template<typename Self>
  constexpr operator decltype(std::declval<Self>().f_())(this Self&& self)
    noexcept(noexcept(std::forward<Self>(self).f_()))
  {
    return std::forward<Self>(self).f_();
  }
private:
  Function f_;
};
```

File Descriptor

`elide` plumbs the return value from `std::execution::connect` directly into the storage of the `std::optional` with no copies or moves.

```
class file_descriptor : public ::exec::safe_file_descriptor {
  //  ...
  void start_() noexcept {
    stop_.emplace();
    if (read_)
      read_->acquire();
    if (write_)
      write_->acquire();
    epoll::events_type events = 0;
    if (read_)
      events |= EPOLLIN;
    if (write_)
      events |= EPOLLOUT;
    op_.emplace(
      elide(
        [&]() noexcept {
          return std::execution::connect(
            ep_.wait(native_handle(), events),
            receiver_(*this));
        }));
    std::execution::start(*op_);
  }
  /*  ...
   */
};
```

LSEG

File Descriptor

The stop token from each child operation will be wired to this callback which will propagate all stop signals to the single stop source being used for the parent operation.

```
class file_descriptor : public ::exec::safe_file_descriptor {
  //  ...
  struct on_stop_request_ {
    explicit on_stop_request_(file_descriptor& self) noexcept
      : self_(self)
    {}
    void operator()() && noexcept {
      self_.stop_->request_stop();
    }
  private:
    file_descriptor& self_;
  };
  /*  ...
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   */
};
```

LSEG

File Descriptor

Completion signals are simply passed through.

```
class file_descriptor : public ::exec::safe_file_descriptor {
  //  ...
  template<bool Readable, typename Receiver>
  class operation_state_ : base_ {
    file_descriptor& self_;
    Receiver receiver_;
    void value() noexcept final {
      std::execution::set_value(std::move(receiver_));
    }
    void error(std::exception_ptr ex) noexcept final {
      std::execution::set_error(
        std::move(receiver_),
        std::move(ex));
    }
    void stop() noexcept final {
      std::execution::set_stopped(std::move(receiver_));
    }
    /*  ...
     *
     *
     *
     */
  };
  //  ...
};
```

LSEG

File Descriptor

Storing the stop callback.

```
class file_descriptor : public ::exec::safe_file_descriptor {
  //  ...
  template<bool Readable, typename Receiver>
  class operation_state_ : base_ {
    //  ...
    using stop_callback_ = std::stop_callback_for_t<
      std::stop_token_of_t<
        std::execution::env_of_t<
          Receiver>>,
      on_stop_request_>;
    std::optional<stop_callback_> callback_;
    /*  ...
     *
     *
     *
     *
     *
     *
     *
     *
     *
     *
     */
  };
  //  ...
};
```

LSEG

File Descriptor

Type erasure of operations against the stop token and callback.

```
class file_descriptor : public ::exec::safe_file_descriptor {
  //  ...
  template<bool Readable, typename Receiver>
  class operation_state_ : base_ {
    //  ...
    bool stop_requested() const noexcept final {
      return std::get_stop_token(
        std::execution::get_env(
          receiver_)).stop_requested();
    }
    void release() noexcept final {
      callback_.reset();
    }
    void acquire() noexcept final {
      callback_.emplace(
        std::get_stop_token(
          std::execution::get_env(receiver_)),
        on_stop_request_(self_));
    }
    /*  ...
     *
     */
  };
  //  ...
};
```

LSEG

File Descriptor

Opt in concept and constructor.

```
class file_descriptor : public ::exec::safe_file_descriptor {
  //  ...
  template<bool Readable, typename Receiver>
  class operation_state_ : base_ {
    //  ...
  public:
    using operation_state_concept =
      std::execution::operation_state_t;
    explicit operation_state_(
      file_descriptor& self,
      Receiver receiver) noexcept(
        std::is_nothrow_move_constructible_v<Receiver>)
        : self_(self),
          receiver_(std::move(receiver))
    {}
    /*  ...
     *
     *
     *
     *
     *
     */
  };
  //  ...
};
```

LSEG

File Descriptor

Starting a sub operation starts the parent operation if there is no parent operation, otherwise it requests that the parent stop and indicates that the interest set should be updated.

```
class file_descriptor : public ::exec::safe_file_descriptor {
  //  ...
  template<bool Readable, typename Receiver>
  class operation_state_ : base_ {
    //  ...
    void start() & noexcept {
      if constexpr (Readable) {
        self_.read_ = this;
      } else {
        self_.write_ = this;
      }
      if (self_.op_) {
        self_.stop_->request_stop();
        return;
      }
      self_.start_();
    }
  };
  /*  ...
   *
   *
   *
   *
   */
};
```

LSEG

File Descriptor

This is all boilerplate.

```
class file_descriptor : public ::exec::safe_file_descriptor {
  //  ...
  template<bool Readable>
  struct sender_ {
    using sender_concept = std::execution::sender_t;
    using completion_signatures =
      std::execution::completion_signatures<
        std::execution::set_value_t(),
        std::execution::set_error_t(std::exception_ptr),
        std::execution::set_stopped_t()>;
    explicit sender_(file_descriptor& self) noexcept
      : self_(self)
    {}
    template<
      std::execution::receiver_of<completion_signatures> Receiver>
    auto connect(Receiver receiver) const noexcept(/* ... */) {
      return operation_state_<Readable, Receiver>(
        self_,
        std::move(receiver));
    }
  private:
    file_descriptor& self_;
  };
  //  ...
};
```

LSEG

File Descriptor

This is also boilerplate.

```
class file_descriptor : public ::exec::safe_file_descriptor {
  //  ...
public:
  auto wait_readable() noexcept {
    return sender_<true>(*this);
  }
  auto wait_writable() noexcept {
    return sender_<false>(*this);
  }
private:
  /*  ...
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   */
};
```

LSEG

File Descriptor

This environment provides the appropriate stop token to the operation against the `epoll`.

```
class file_descriptor : public ::exec::safe_file_descriptor {
  //  ...
  struct env_ {
    explicit env_(file_descriptor& self) noexcept
      : self_(self)
    {}
    std::inplace_stop_token query(const std::get_stop_token_t&)
      const noexcept
    {
      return self_.stop_->get_token();
    }
  private:
    file_descriptor& self_;
  };
  /*  ...
   *
   *
   *
   *
   *
   *
   *
   *
   *
   */
};
```

LSEG

File Descriptor

Note `get_env` which completes integration of the stop source into this receiver (and the operation to which it is passed).

```
class file_descriptor : public ::exec::safe_file_descriptor {
  //  ...
  class receiver_ {
    file_descriptor& self_;
  public:
    using receiver_concept = std::execution::receiver_t;
    explicit receiver_(file_descriptor& self) noexcept
      : self_(self)
    {}
    env_ get_env() const noexcept {
      return env_(self_);
    }
    /*  ...
     *
     *
     *
     *
     *
     *
     *
     *
     *
     */
  };
  //  ...
};
```

LSEG

File Descriptor

Before taking further action perform clean up.

```
class file_descriptor : public ::exec::safe_file_descriptor {
  //  ...
  class receiver_ {
    //  ...
    void set_value(const epoll::events_type events) && noexcept {
      self_.begin_completion_();
    }
    //  ...
  };
  //  ...
};
```

LSEG

File Descriptor

Depending on whether `EPOLLIN`, `EPOLLOUT`, or both is indicated prepare sub operation for completion.

```
class file_descriptor : public ::exec::safe_file_descriptor {
  //  ...
  class receiver_ {
    //  ...
    void set_value(const epoll::events_type events) && noexcept {
      self_.begin_completion_();
      const auto read =
        (EPOLLIN & events)
          ? std::exchange(self_.read_, nullptr)
          : nullptr;
      const auto write =
        (EPOLLOUT & events)
          ? std::exchange(self_.write_, nullptr)
          : nullptr;
    }
    //  ...
  };
  //  ...
};
```

LSEG

File Descriptor

If there are remaining sub operations restart the wait operation.

```
class file_descriptor : public ::exec::safe_file_descriptor {
  //  ...
  class receiver_ {
    //  ...
    void set_value(const epoll::events_type events) && noexcept {
      self_.begin_completion_();
      const auto read =
        (EPOLLIN & events)
          ? std::exchange(self_.read_, nullptr)
          : nullptr;
      const auto write =
        (EPOLLOUT & events)
          ? std::exchange(self_.write_, nullptr)
          : nullptr;
      if (self_.read_ || self_.write_)
        self_.start_();



    }
    //  ...
  };
  //  ...
};
```

LSEG

File Descriptor

Send completion signal(s).

```
class file_descriptor : public ::exec::safe_file_descriptor {
  //  ...
  class receiver_ {
    //  ...
    void set_value(const epoll::events_type events) && noexcept {
      self_.begin_completion_();
      const auto read =
        (EPOLLIN & events)
          ? std::exchange(self_.read_, nullptr)
          : nullptr;
      const auto write =
        (EPOLLOUT & events)
          ? std::exchange(self_.write_, nullptr)
          : nullptr;
      if (self_.read_ || self_.write_)
        self_.start_();
      if (read)
        read->value();
      if (write)
        write->value();
    }
    //  ...
  };
  //  ...
};
```

LSEG

File Descriptor

Error completion fans out to both sub operations.

```
class file_descriptor : public ::exec::safe_file_descriptor {
  //  ...
  class receiver_ {
    //  ...
    void set_error(std::exception_ptr ex) && noexcept {
      self_.begin_completion_();
      const auto read = std::exchange(self_.read_, nullptr);
      const auto write = std::exchange(self_.write_, nullptr);
      if (read) {
        read->error(ex);
      }
      if (write) {
        write->error(std::move(ex));
      }
    }
    /*  ...
     *
     *
     *
     *
     *
     */
  };
  //  ...
};
```

LSEG

File Descriptor

If either sub operation requested a stop, it's stopped.

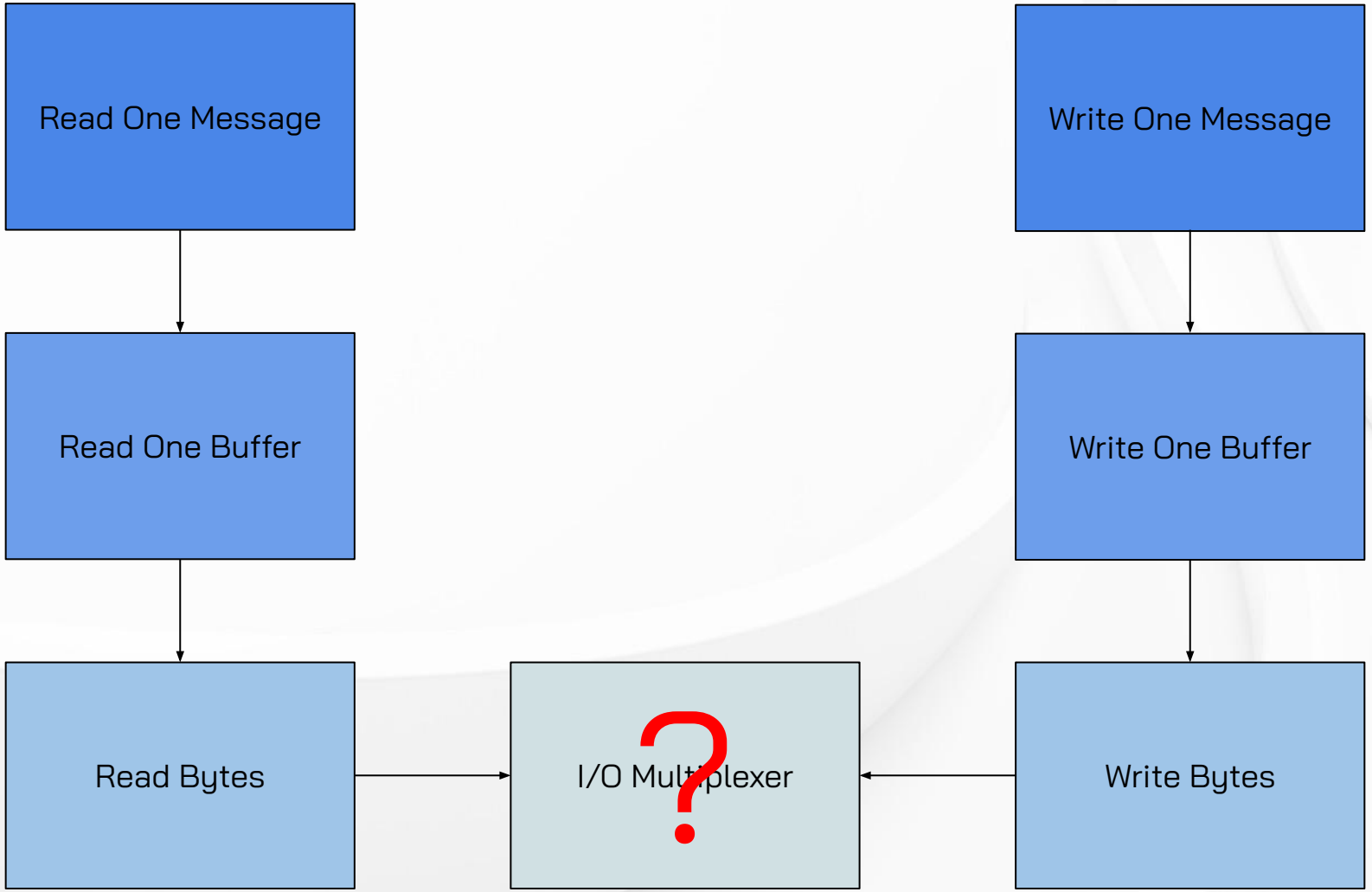
Otherwise if there are still outstanding operations we restart (this starts a new operation with the correct interest set).

```
class file_descriptor : public ::exec::safe_file_descriptor {
  //  ...
  class receiver_ {
    //  ...
    void set_stopped() && noexcept {
      self_.begin_completion_();
      const auto read =
        (self_.read_ && self_.read_->stop_requested())
          ? std::exchange(self_.read_, nullptr)
          : nullptr;
      const auto write =
        (self_.write_ && self_.write_->stop_requested())
          ? std::exchange(self_.write_, nullptr)
          : nullptr;
      if (self_.read_ || self_.write_)
        self_.start_();
      if (read)
        read->stop();
      if (write)
        write->stop();
    }
  };
};
```

LSEG

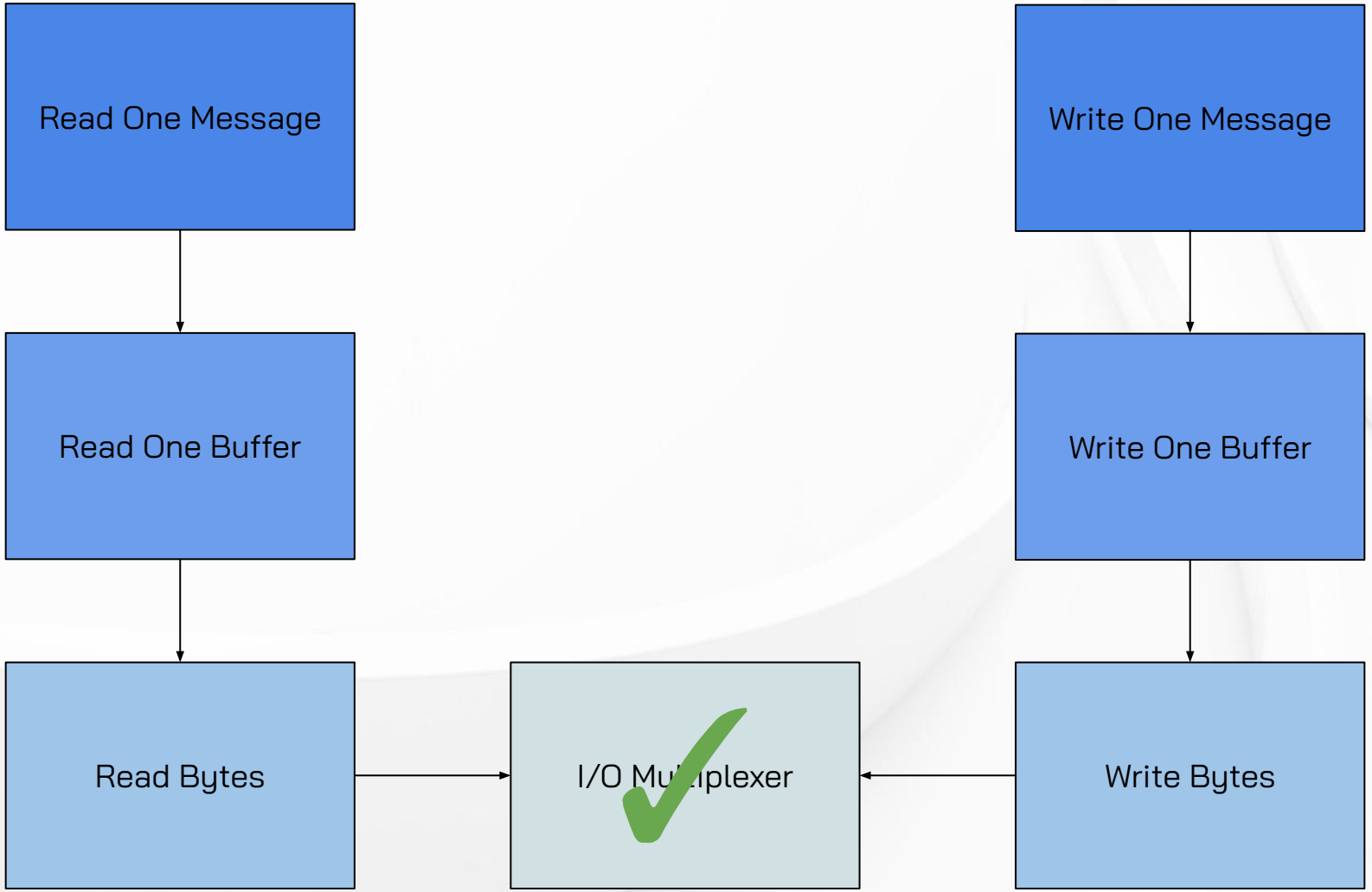
Where to Start?

Filling in the Blank Slate



Where to Start?

Filling in the Blank Slate



Additional Operations

Checking for readability and writability isn't a sufficient basis operation: We need to actually read, actually write, and actually accept (note that accepting sockets upon which accepting will not block report as readable).

```
struct file_descriptor : ::exec::safe_file_descriptor {
  //  ...
  std::execution::sender auto read(
    std::span<std::byte> span) noexcept;
  std::execution::sender auto write(
    std::span<const std::byte> span) noexcept;
  std::execution::sender auto accept() noexcept;
};
```

Asynchronous Read

`f` actually performs the read, but how do we asynchronously get it started?

```
auto read(std::span<std::byte> span) noexcept {
  const auto f = [span, fd = native_handle()]() {
    const auto res = ::read(fd, span.data(), span.size());
    if (res == -1) {
      throw system_error("read");
    }
    return std::size_t(res);
  };

}
```

LSEG

Asynchronous Read

Surely `wait_readable` is going to be involved, but how do we compose the two?

```
auto read(std::span<std::byte> span) noexcept {
  const auto f = [span, fd = native_handle()]() {
    const auto res = ::read(fd, span.data(), span.size());
    if (res == -1) {
      throw system_error("read");
    }
    return std::size_t(res);
  };
        wait_readable()
}
```

LSEG

Asynchronous Read

All we need is `std::execution::then`.

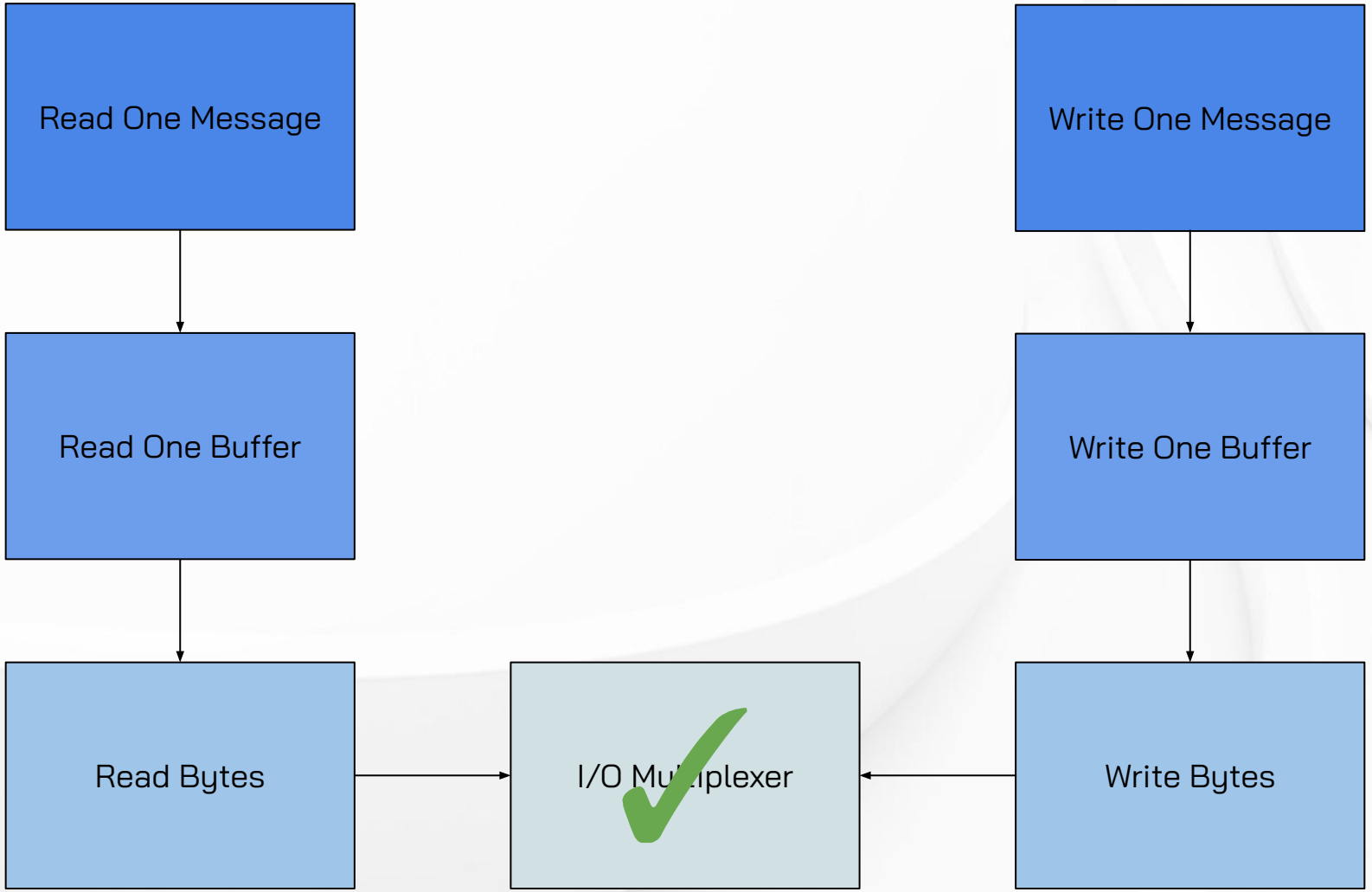
This is a fully-working implementation, with stop support and error handling which cascade downstream implicitly.

```
auto read(std::span<std::byte> span) noexcept {
  const auto f = [span, fd = native_handle()]() {
    const auto res = ::read(fd, span.data(), span.size());
    if (res == -1) {
      throw system_error("read");
    }
    return std::size_t(res);
  };
  return wait_readable() | std::execution::then(f);
}
```

LSEG

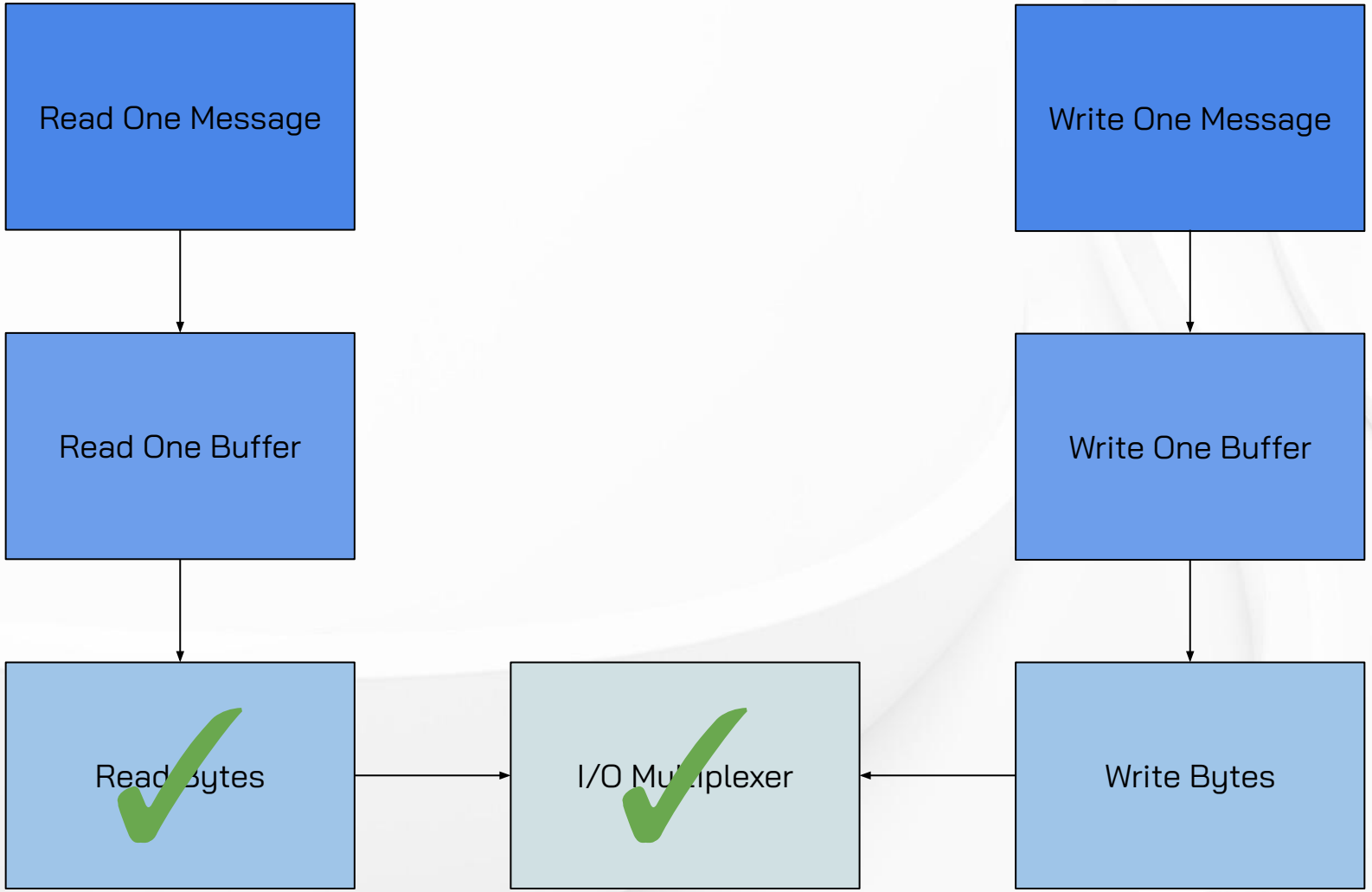
Where to Start?

Filling in the Blank Slate



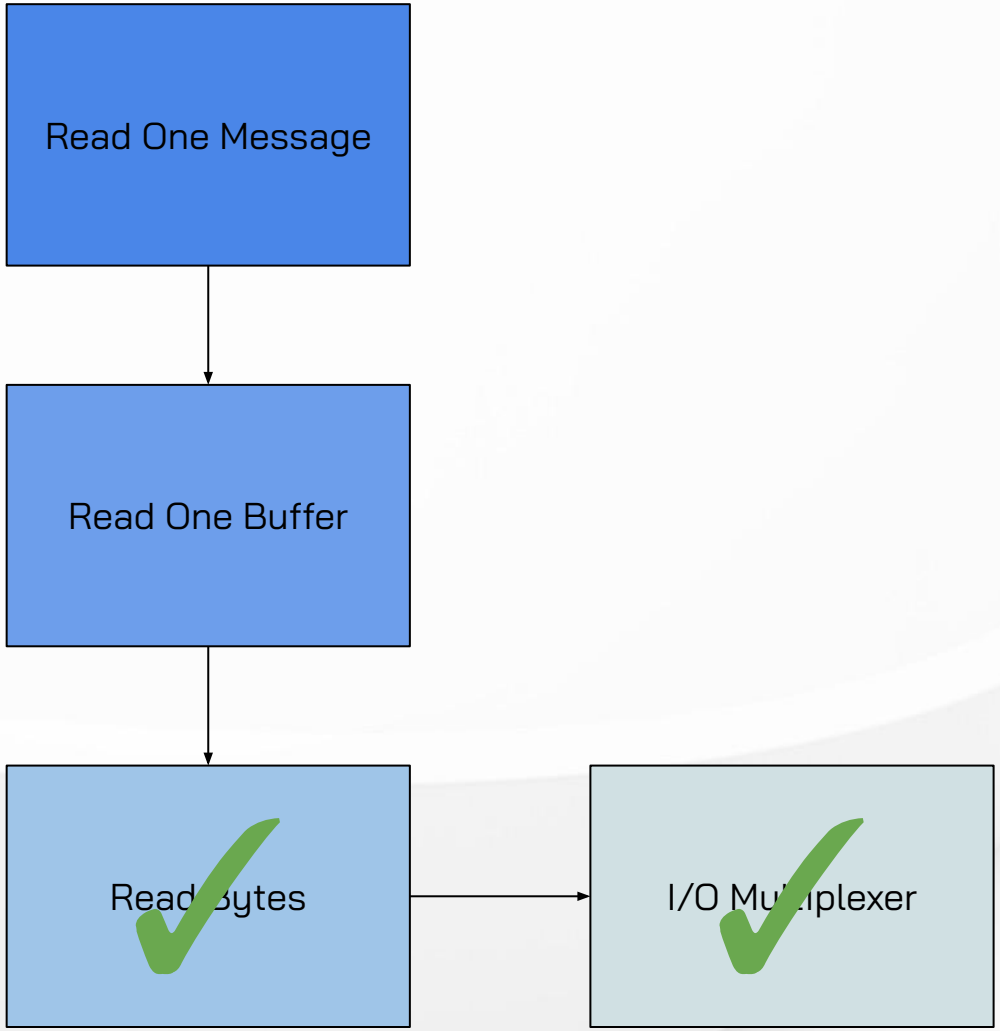
Where to Start?

Filling in the Blank Slate



Where to Start?

Filling in the Blank Slate



Read

Generic operation, a free function.

```
template<typename Readable>
auto read(Readable& readable, const std::span<std::byte> span) {









}
```

LSEG

Read

Synchronous version.

```
template<typename Readable>
void read(Readable& readable, std::span<std::byte> span) {




}
```

Read

Loop until `span` is emptied.

```
template<typename Readable>
void read(Readable& readable, std::span<std::byte> span) {
  do {



  } while (!span.empty());
}
```

Read

Perform read (imagine it's synchronous).

```
template<typename Readable>
void read(Readable& readable, std::span<std::byte> span) {
  do {
    const auto read = readable.read(span);



  } while (!span.empty());
}
```

LSEG

Read

Handle EOF.

```
template<typename Readable>
void read(Readable& readable, std::span<std::byte> span) {
  do {
    const auto read = readable.read(span);
    if (!read) {
      throw eof_error{};
    }

  } while (!span.empty());
}
```

Read

Remove populated prefix from span.

```
template<typename Readable>
void read(Readable& readable, std::span<std::byte> span) {
  do {
    const auto read = readable.read(span);
    if (!read) {
      throw eof_error{};
    }
    span = span.subspan(read);
  } while (!span.empty());
}
```

Read

Currying `readable.read`. But does this work?

```
template<typename Readable>
void read(Readable& readable, std::span<std::byte> span) {
  const auto impl = [&, span]() {
    return readable.read(span);
  };
  do {
    const auto read = impl();
    if (!read) {
      throw eof_error{};
    }
    span = span.subspan(read);
  } while (!span.empty());
}
```

LSEG

Read

Extracting the loop body into a function.

```
template<typename Readable>
void read(Readable& readable, std::span<std::byte> span) {
  const auto impl = [&]() {
    const auto read = readable.read(span);
    if (!read) {
      throw eof_error{};
    }
    span = span.subspan(read);
    return span.empty();
  };
  while (!impl());
}
```

LSEG

Read

Notice the argument is used as mutable state.

```
template<typename Readable>
void read(Readable& readable, std::span<std::byte> span) {
  const auto impl = [&]() {
    const auto read = readable.read(span);
    if (!read) {
      throw eof_error{};
    }
    span = span.subspan(read);
    return span.empty();
  };
  while (!impl());
}
```

LSEG

Read

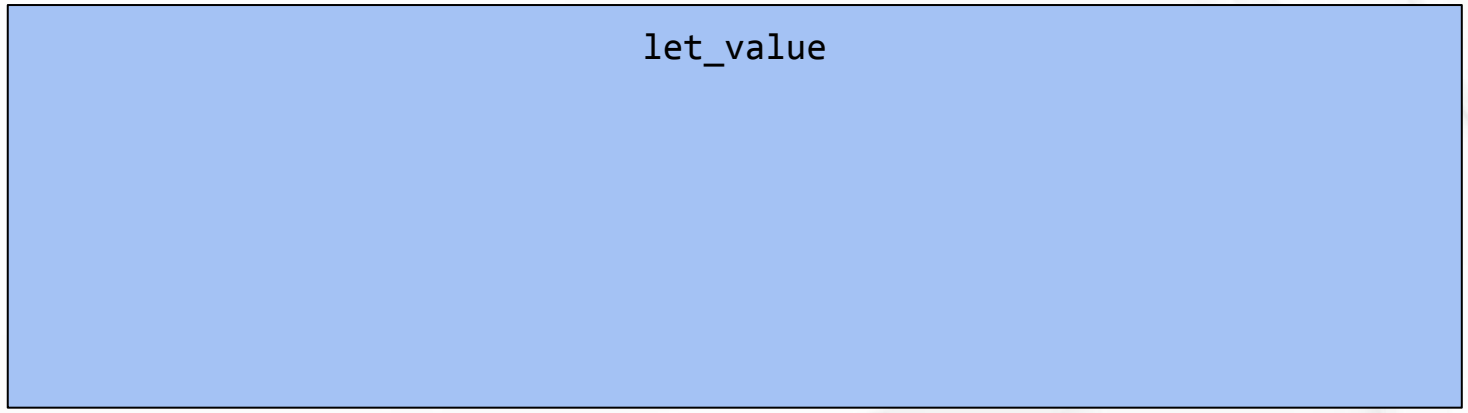
Why `std::execution::just` into `std::execution::let_value`?

```
template<typename Readable>
auto read(Readable& readable, const std::span<std::byte> span) {
  return
    std::execution::just(span) |
    std::execution::let_value([&readable](
      std::span<std::byte>& span)
    {




    });
}
```

LSEG

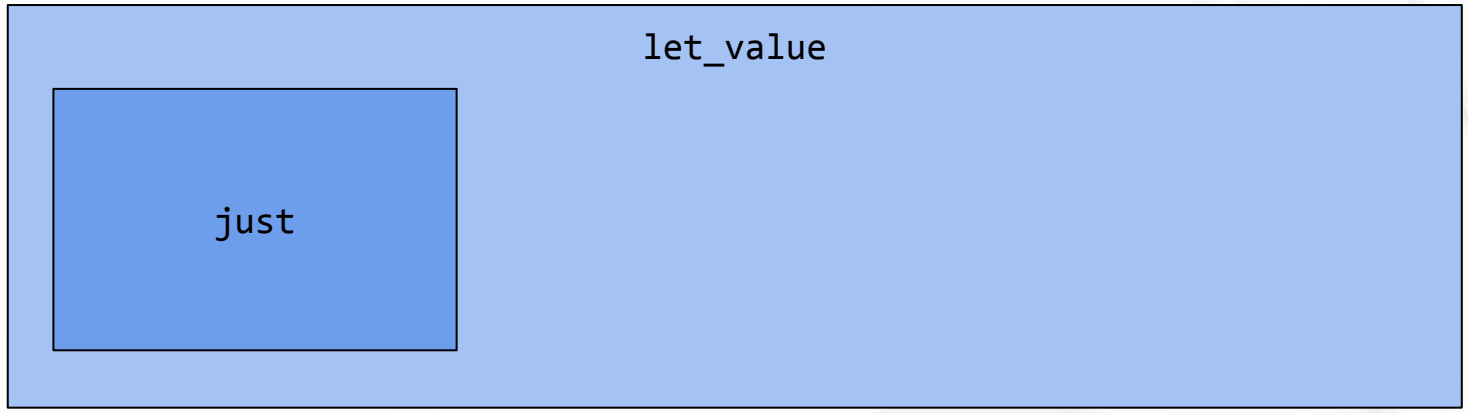
Operation State Nesting

Asynchronous stack frames



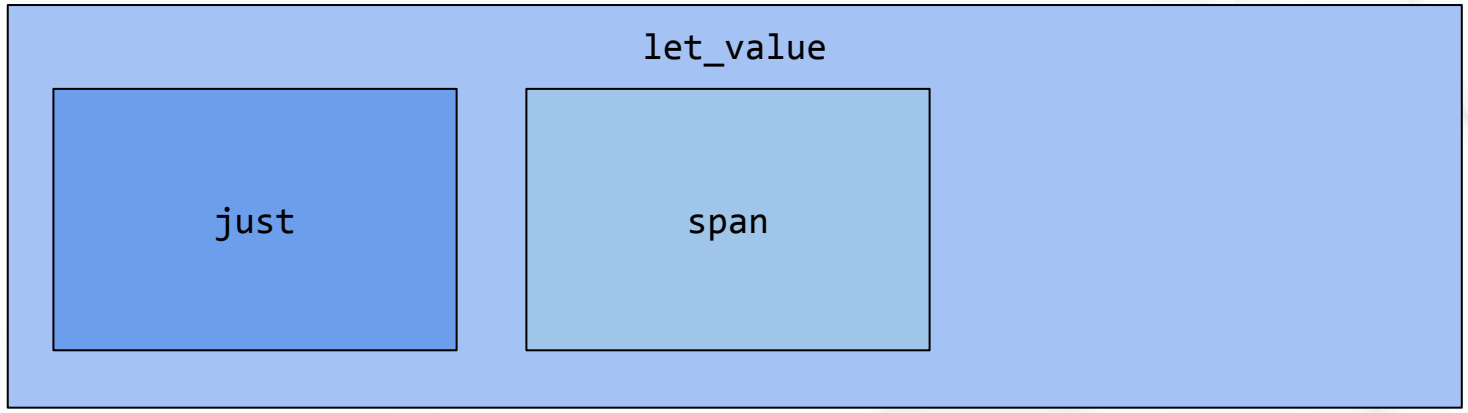
Operation State Nesting

Call predecessor



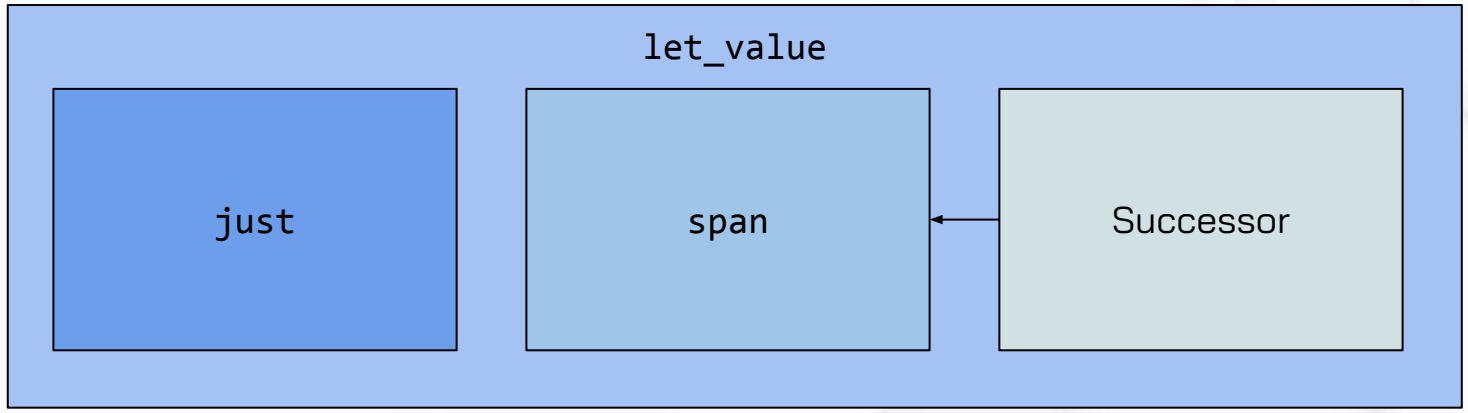
Operation State Nesting

Store return value



Operation State Nesting

Call successor



Read

`exec::repeat_effect_until` is an extension provided by stdexec.

```
template<typename Readable>
auto read(Readable& readable, const std::span<std::byte> span) {
  return
    std::execution::just(span) |
    std::execution::let_value([&readable](
      std::span<std::byte>& span)
    {
      return ::exec::repeat_effect_until(









          }));
    });
}
```

LSEG

Read

Again, why `std::execution::just` into `std::execution::let_value`?

```
template<typename Readable>
auto read(Readable& readable, const std::span<std::byte> span) {
  return
    std::execution::just(span) |
    std::execution::let_value([&readable](
      std::span<std::byte>& span)
    {
      return ::exec::repeat_effect_until(
        std::execution::just() |
        std::execution::let_value([&readable, &span]() {
          return readable.read(span);
        })
      );
    });
}
```

LSEG

Read

Why not this?

```
template<typename Readable>
auto read(Readable& readable, const std::span<std::byte> span) {
  return
    std::execution::just(span) |
    std::execution::let_value([&readable](
      std::span<std::byte>& span)
    {
      return ::exec::repeat_effect_until(
        readable.read(span)
                                          );
    });
}
```

LSEG

Read

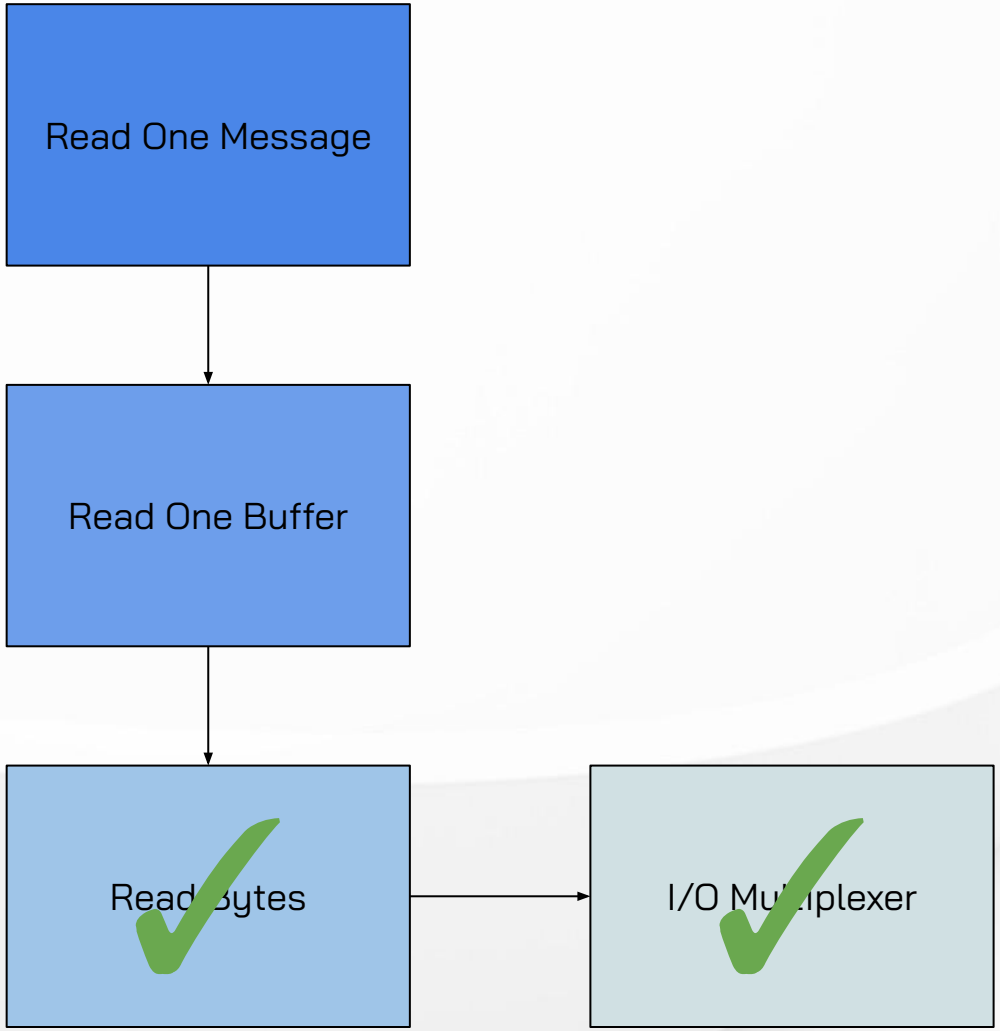
Read bytes and carve them out of the span.

```
template<typename Readable>
auto read(Readable& readable, const std::span<std::byte> span) {
  return
    std::execution::just(span) |
    std::execution::let_value([&readable](
      std::span<std::byte>& span)
    {
      return ::exec::repeat_effect_until(
        std::execution::just() |
        std::execution::let_value([&readable, &span]() {
          return readable.read(span);
        }) |
        std::execution::then([&span](std::size_t bytes) {
          if (!bytes) {
            throw eof_error{};
          }
          span = span.subspan(bytes);
          return span.empty();
        }));
    });
}
```

LSEG

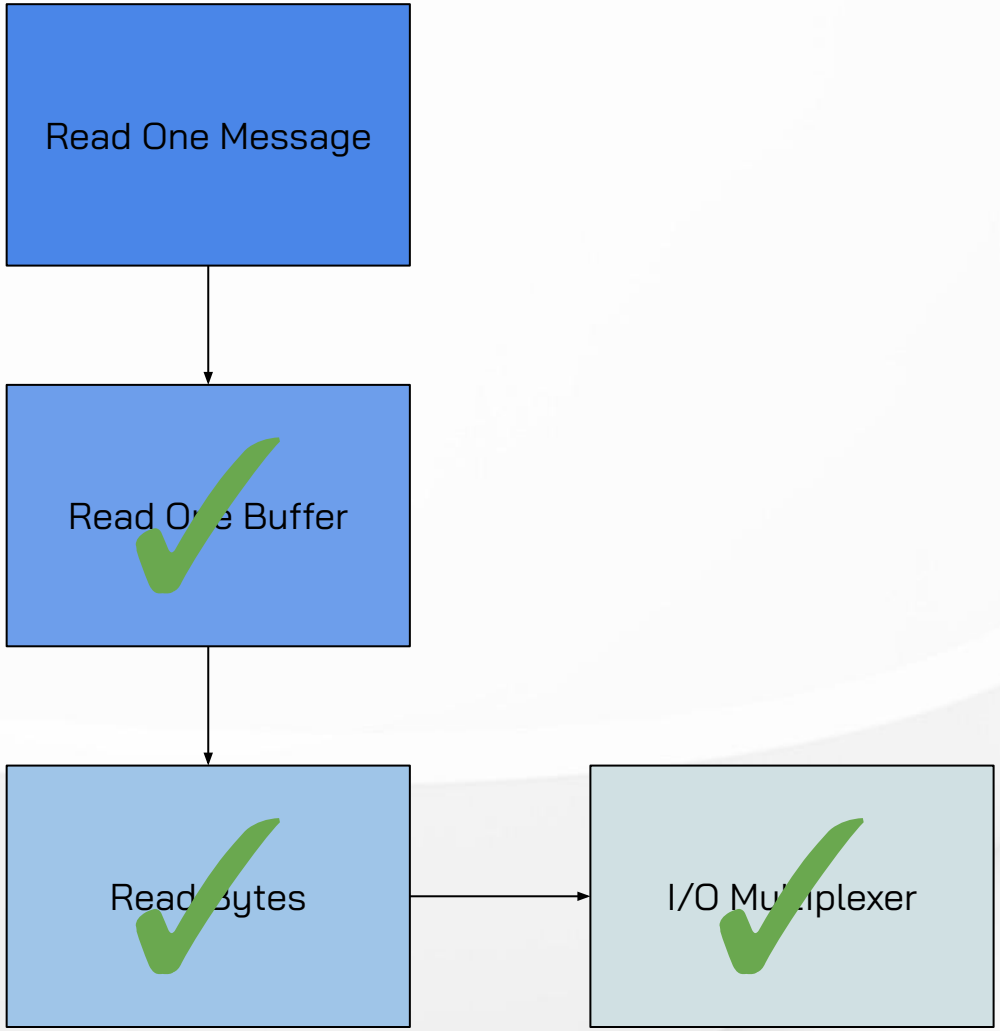
Where to Start?

Filling in the Blank Slate



Where to Start?

Filling in the Blank Slate



DataConn Message Structure

Offset	Field
0	Length (bytes) (little endian)
1	
2	
3	
4	Message type
...	Body (optional)

Reading DataConn

`read_header` reads the DataConn header and generates the size and message type parsed therefrom.

```
template<typename Readable>
std::execution::sender auto read_header(Readable& readable);

template<typename Readable>
auto read(Readable& readable, std::vector<std::byte>& buffer) {









}
```

LSEG

Reading DataConn

After reading the header we need to decide how to proceed, therefore `std::execution::let_value`.

```
template<typename Readable>
std::execution::sender auto read_header(Readable& readable);

template<typename Readable>
auto read(Readable& readable, std::vector<std::byte>& buffer) {
  return
    read_header(readable) |
    std::execution::let_value([&](
      const std::uint32_t size,
      const message_type type)
    {









    });
}
```

LSEG

Reading DataConn

`exec::sequence` starts the first operation, then the second operation, et cetera (it's variadic).

```
template<typename Readable>
std::execution::sender auto read_header(Readable& readable);

template<typename Readable>
auto read(Readable& readable, std::vector<std::byte>& buffer) {
  return
    read_header(readable) |
    std::execution::let_value([&](
      const std::uint32_t size,
      const message_type type)
    {
      buffer.resize(size - 5U);
      auto no_body = std::execution::just(type);
      auto body = ::exec::sequence(
        ::read(readable, buffer),
        no_body);




    });
}
```

LSEG

Reading DataConn

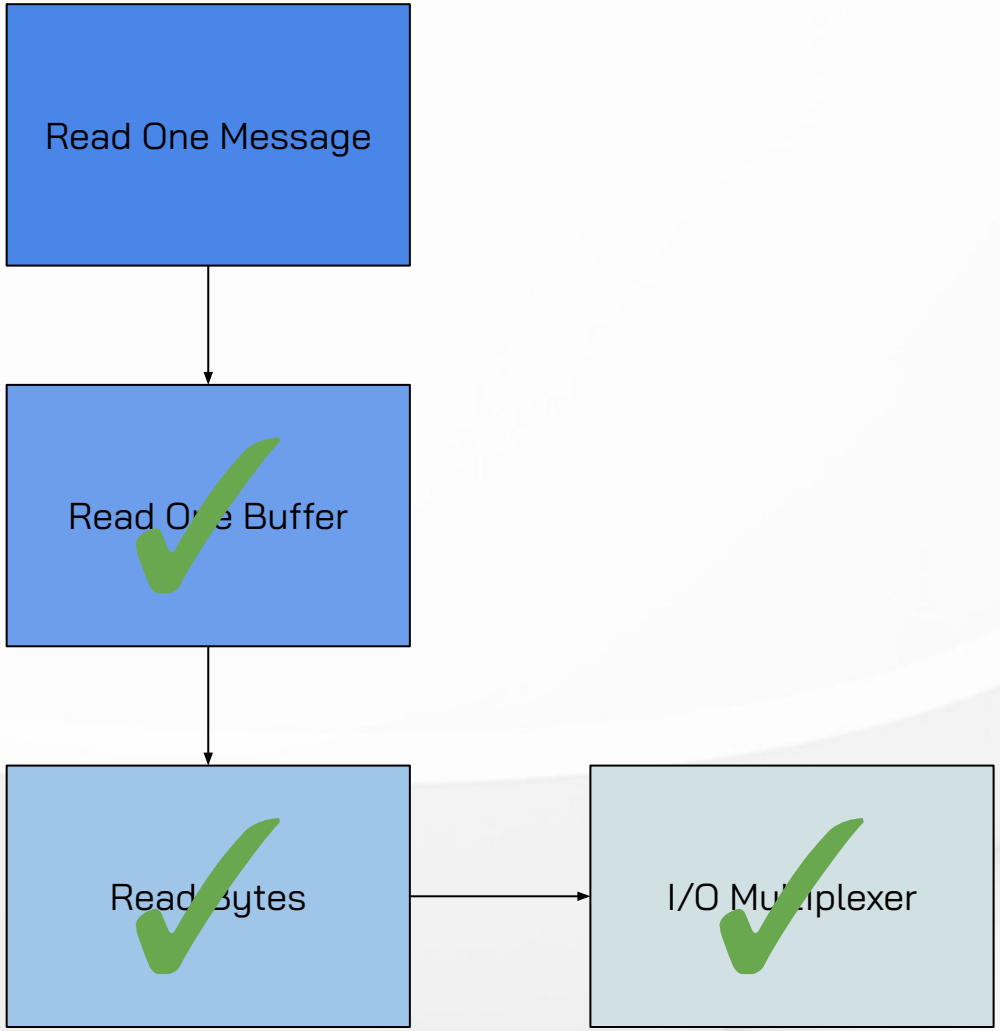
`exec::variant_sender` allows us to implement such a branch in asynchronous execution by allowing us to pick a different sender depending on the situation.

```
template<typename Readable>
std::execution::sender auto read_header(Readable& readable);

template<typename Readable>
auto read(Readable& readable, std::vector<std::byte>& buffer) {
  return
    read_header(readable) |
    std::execution::let_value([&](
      const std::uint32_t size,
      const message_type type)
    {
      buffer.resize(size - 5U);
      auto no_body = std::execution::just(type);
      auto body = ::exec::sequence(
        ::read(readable, buffer),
        no_body);
      using type = ::exec::variant_sender<
        decltype(body),
        decltype(no_body)>;
      if (buffer.empty()) {
        return type(std::move(no_body));
      }
      return type(std::move(body));
    });
}
```

LSEG

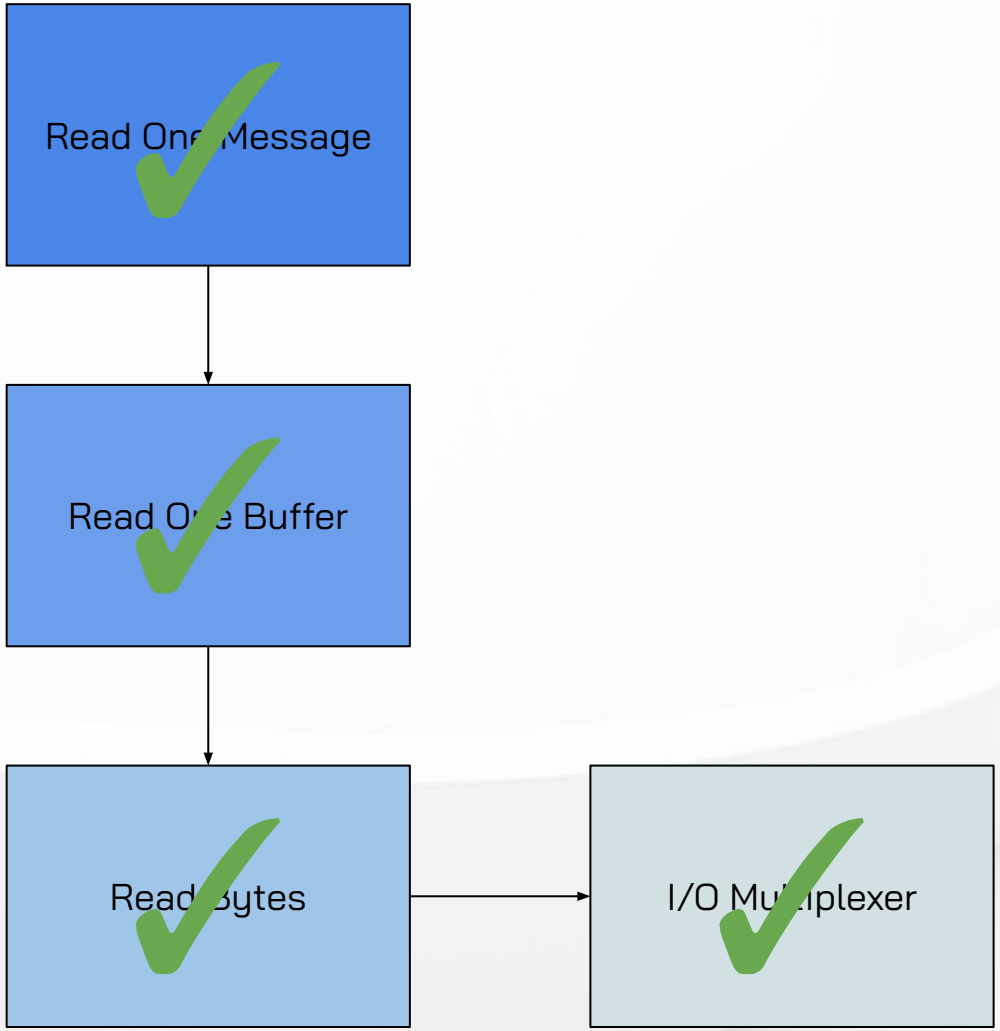
Where to Start?

Filling in the Blank Slate



Where to Start?

Filling in the Blank Slate



DataConn Messages (Client to Server)

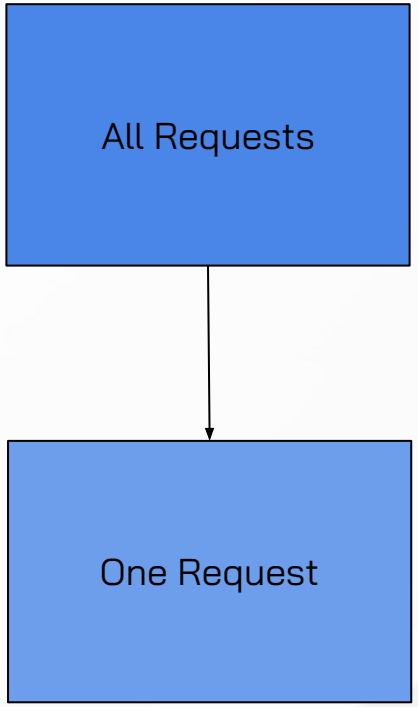
Type	Name	Payload?
R	Request	✓
C	Cancel	

What Next?

Building on Top of Our Primitives

What Next?

Building on Top of Our Primitives

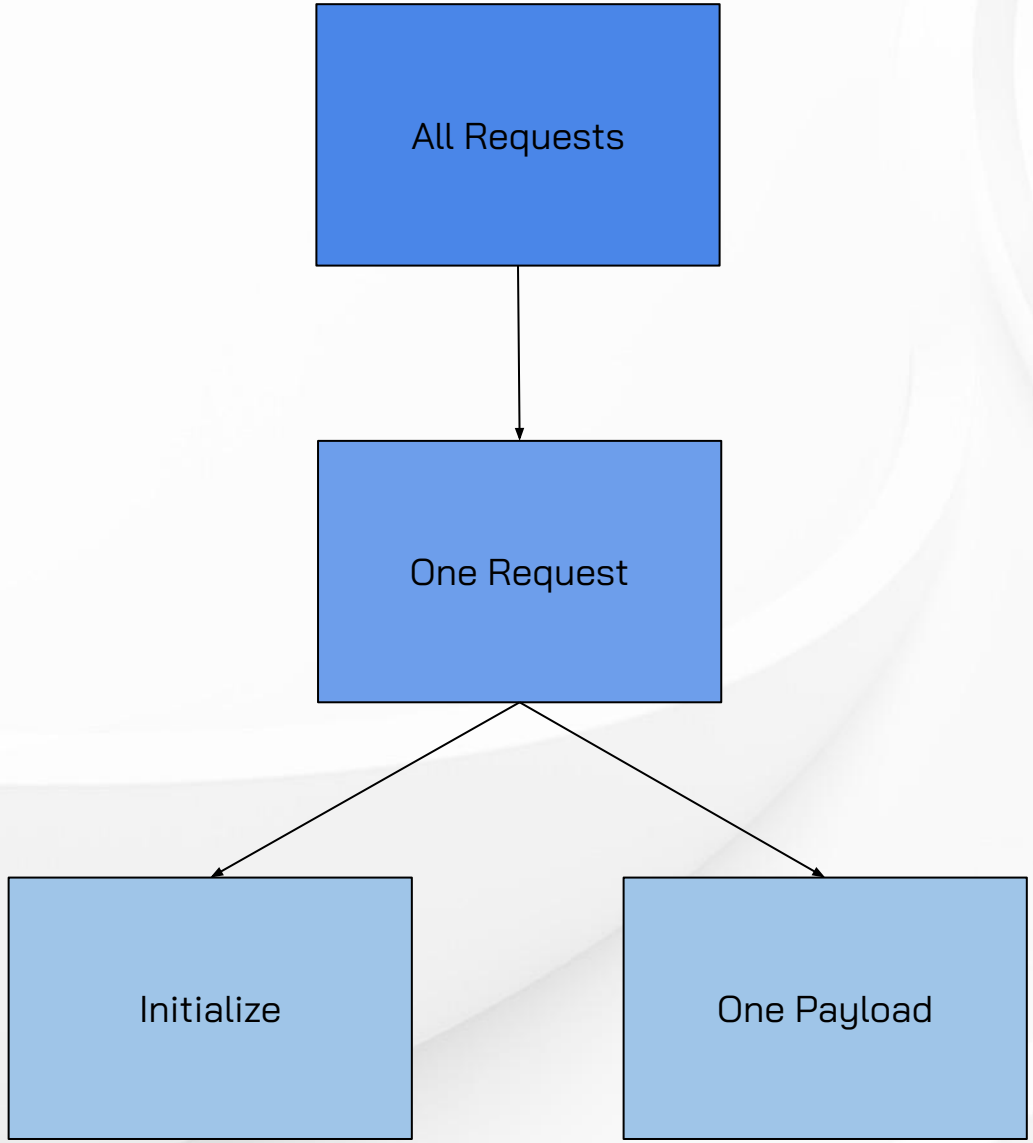


Query

```
enum class query_status { wait, payload, error, complete };

struct /* ... */ {
  template<typename ConstBufferSequence>
  std::variant</* ... */> init(ConstBufferSequence cb);
  query_status run();
  void end() noexcept;
  std::span<std::byte> payload() const noexcept;
  void toggle();
};
```

What Next?

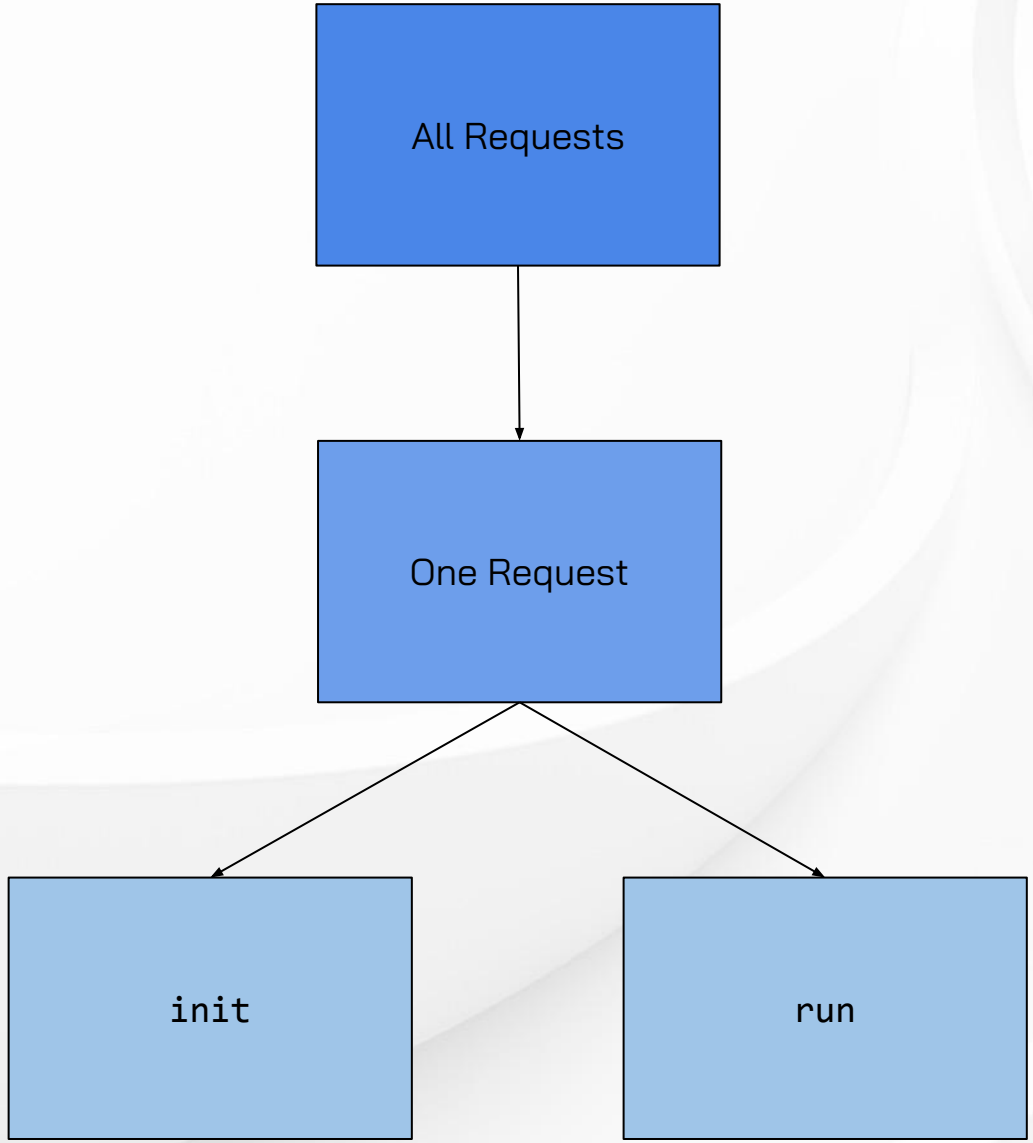
Building on Top of Our Primitives



LSEG

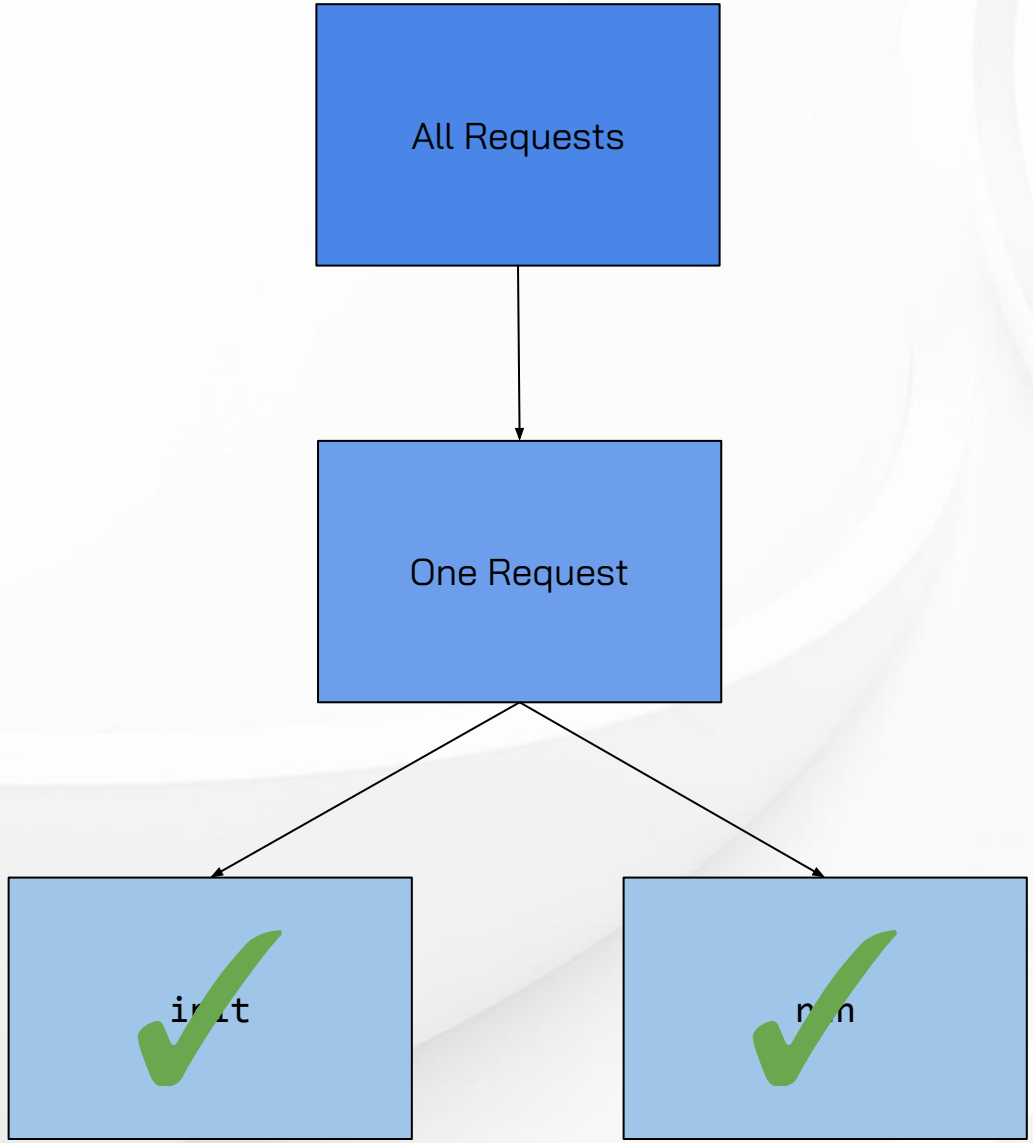
What Next?

Building on Top of Our Primitives



What Next?

Building on Top of Our Primitives



DataConn Messages (Client to Server)

Type	Name	Payload?
R	Request	✓
C	Cancel	

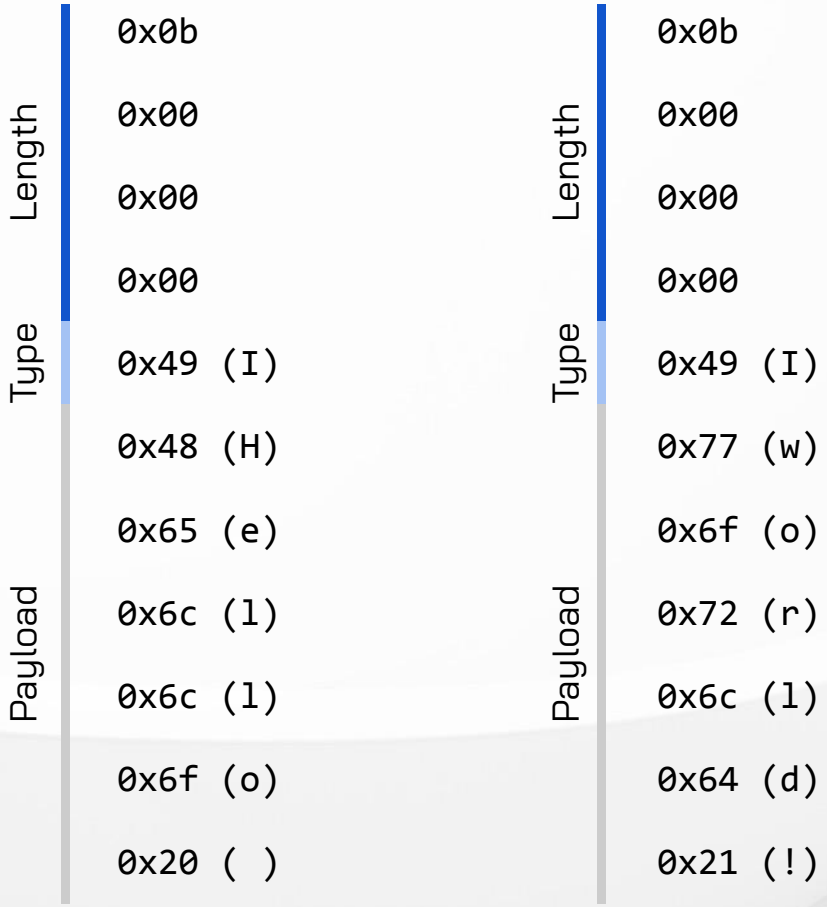
DataConn Messages (Server to Client)

Type	Name	Payload?	Terminal?
A	Acknowledge Request	✓	
J	Reject Request	✓	✓
I	Intermediate Chunk	✓	
F	Final Chunk	✓	✓
E	Error	✓	✓
K	Acknowledge Cancel		✓
H	Heartbeat		

Cancellation vs. Stopping

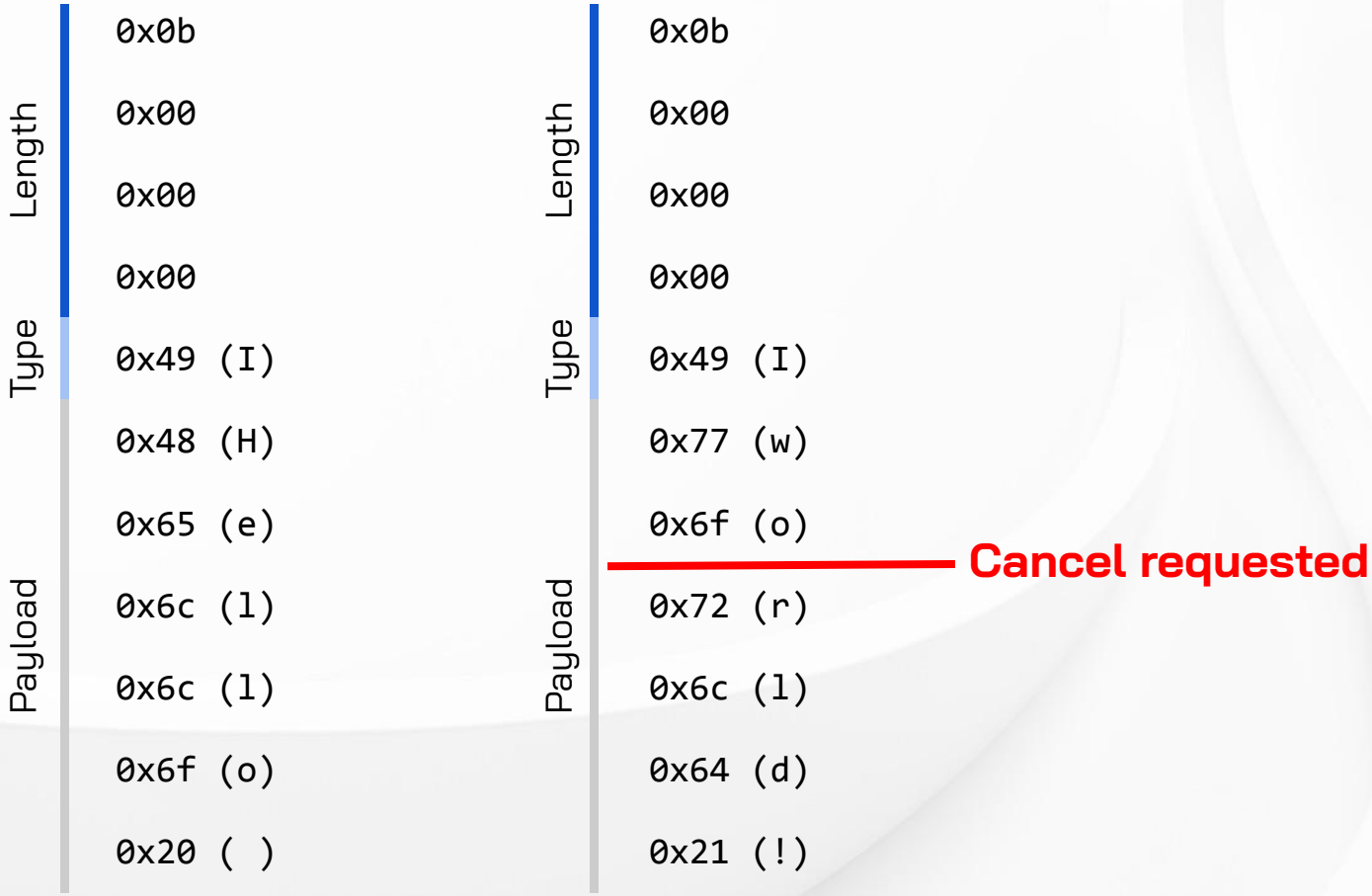
Torn Write

Two Payloads



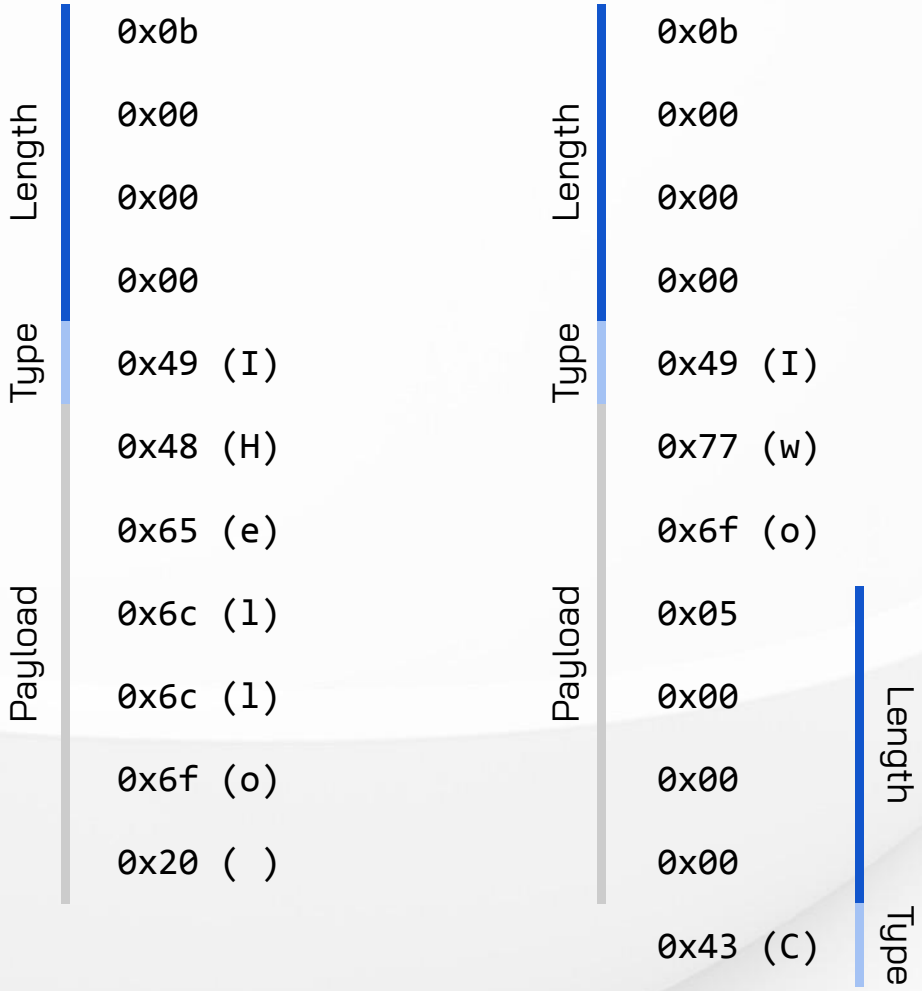
Torn Write

Unfortunately-Timed Cancel Request

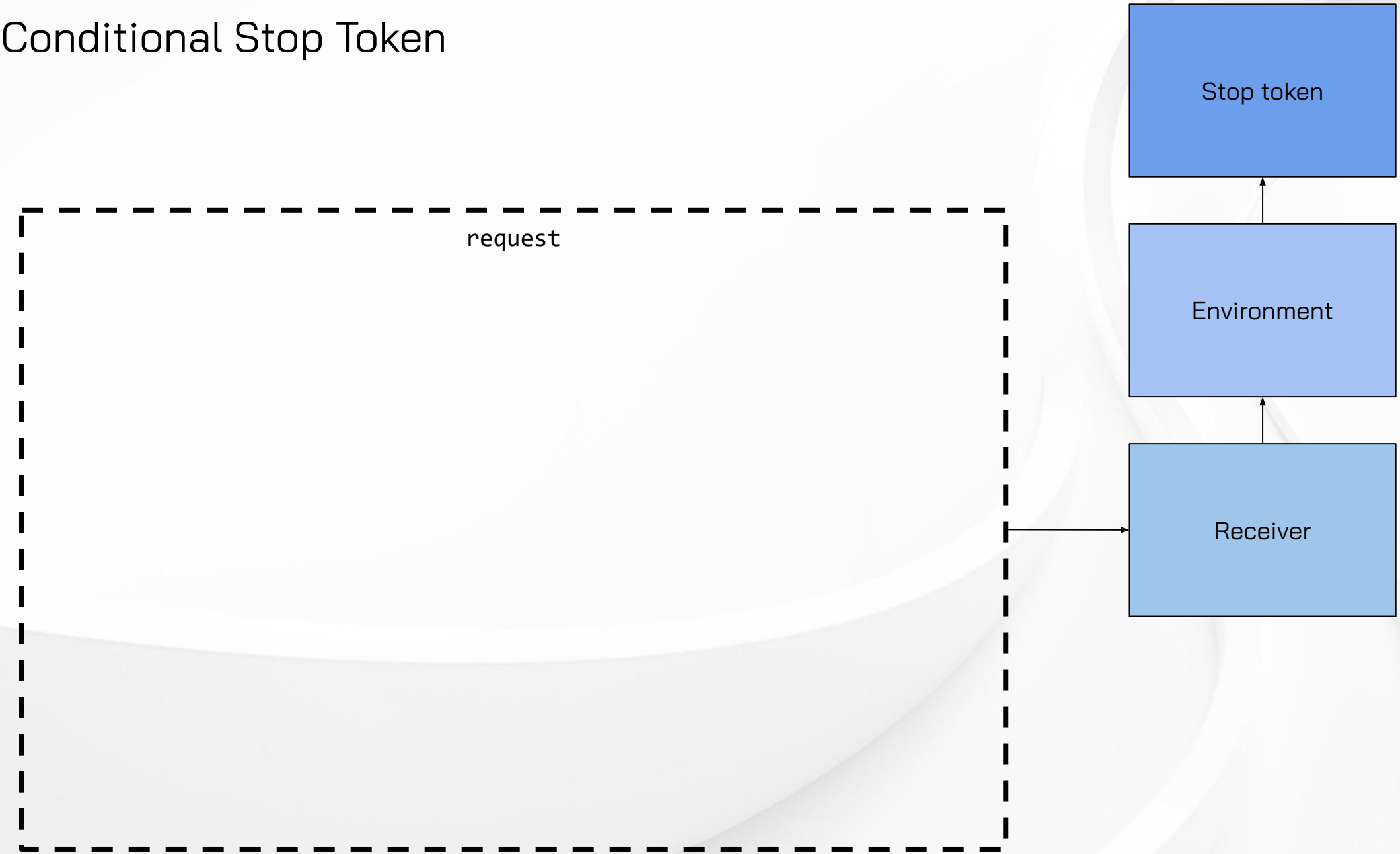


Torn Write

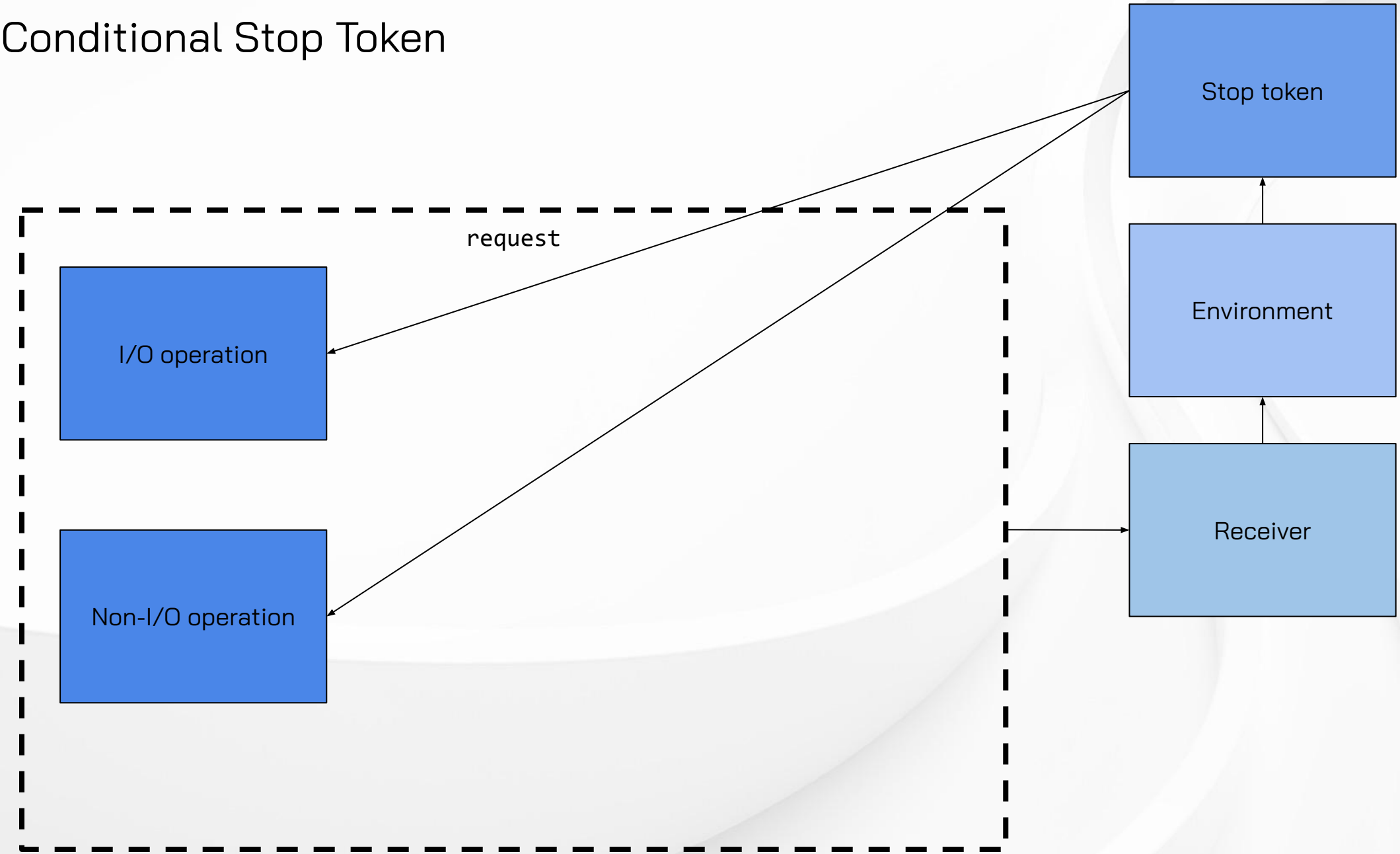
Corrupted Output



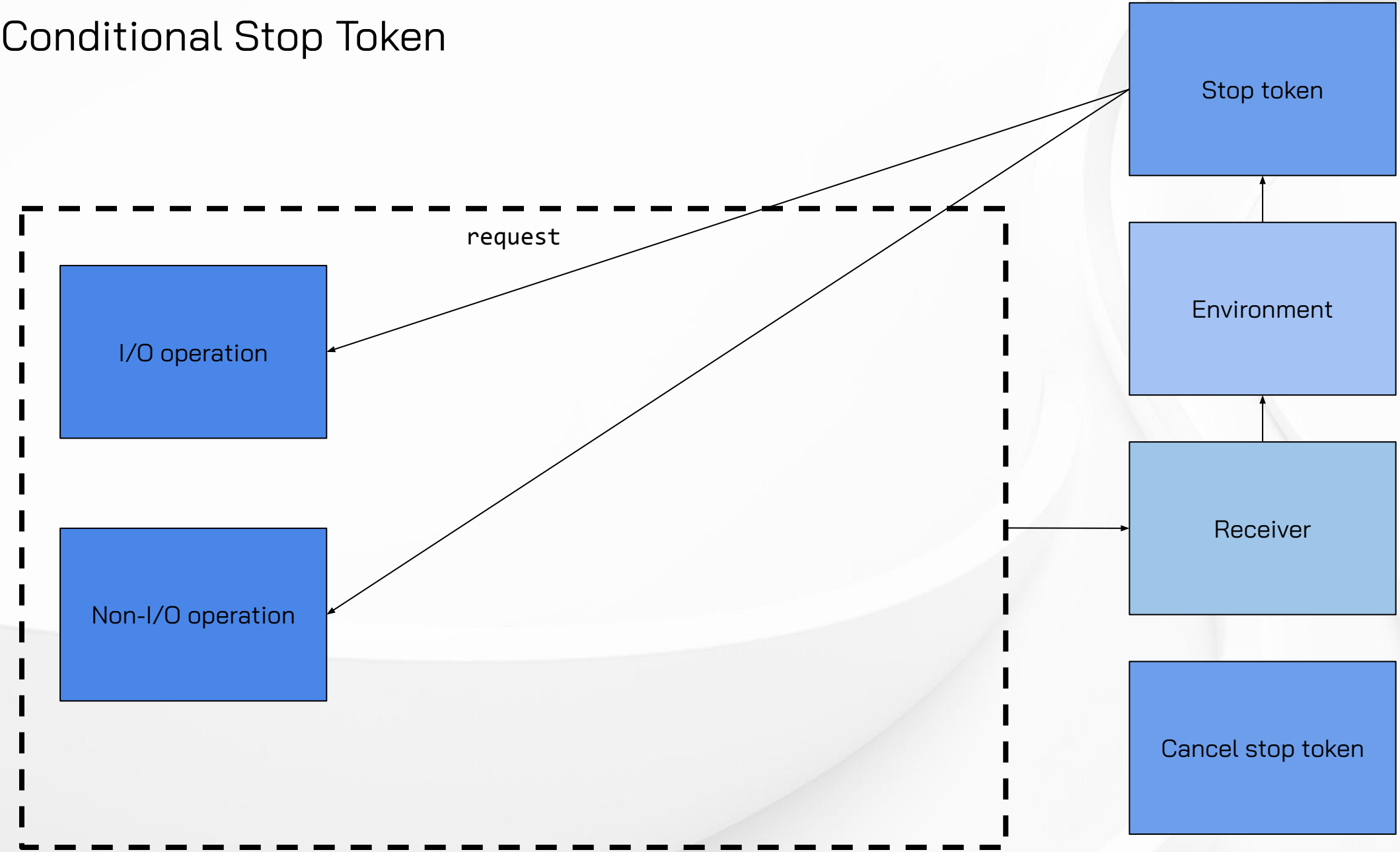
Conditional Stop Token



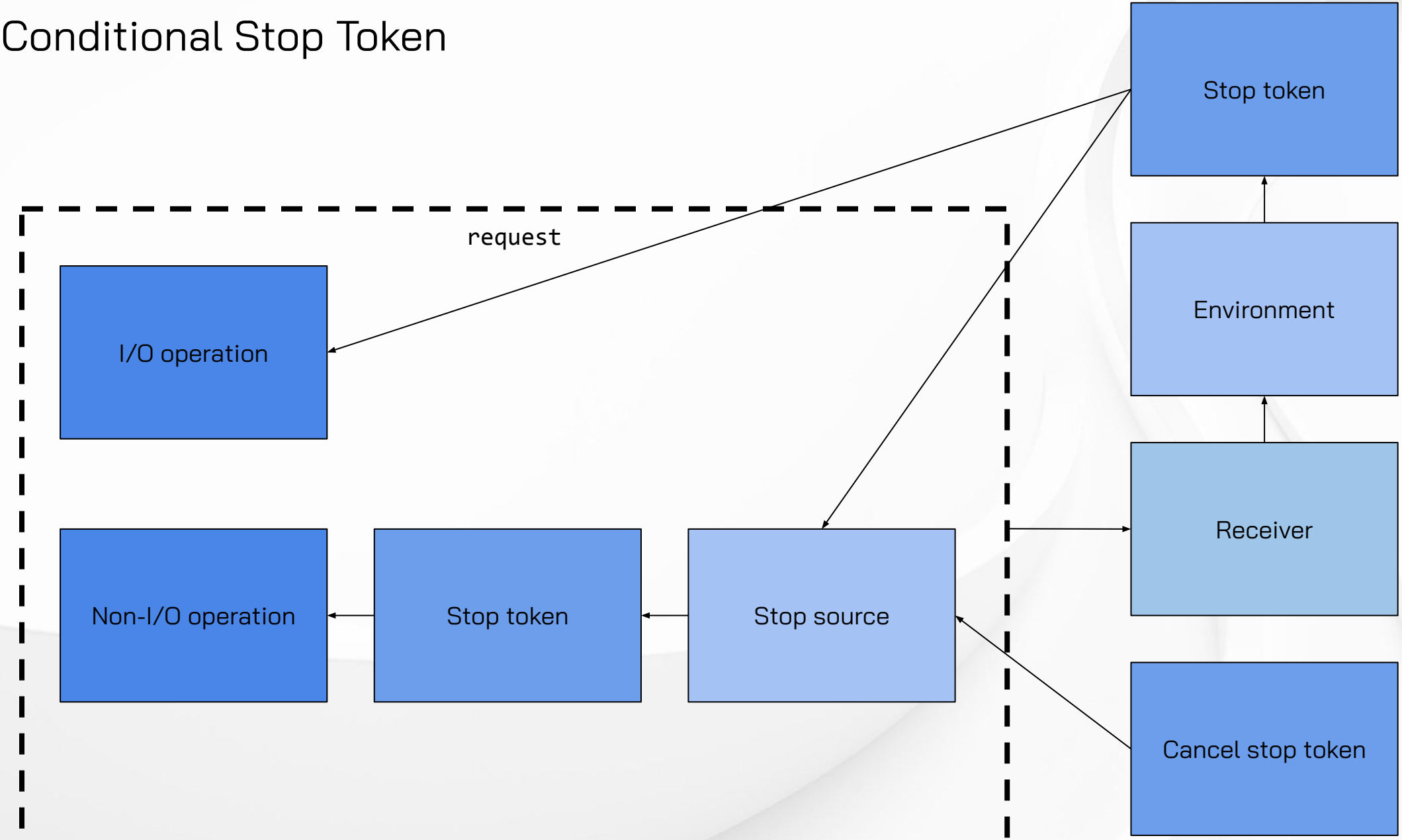
Conditional Stop Token



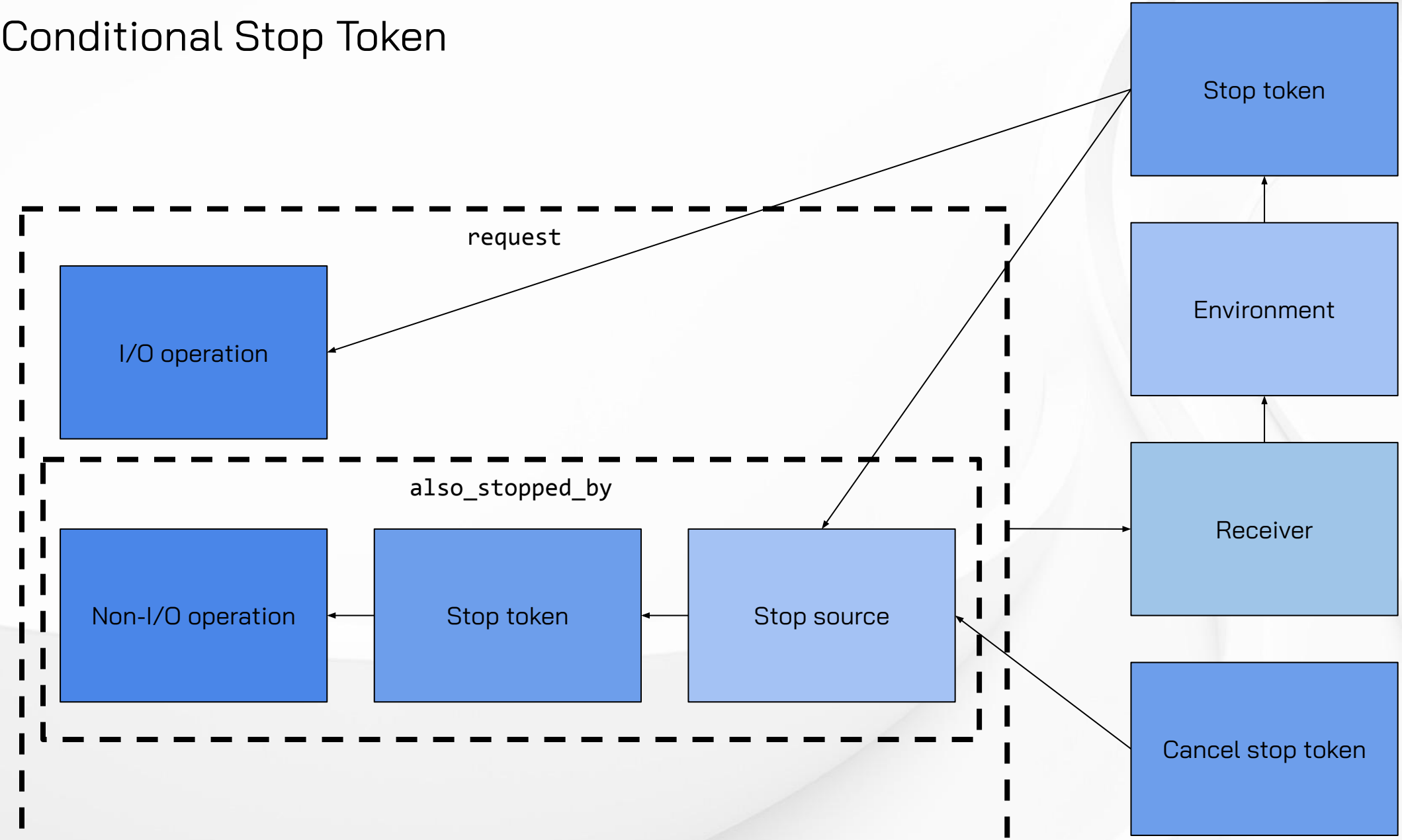
Conditional Stop Token



Conditional Stop Token

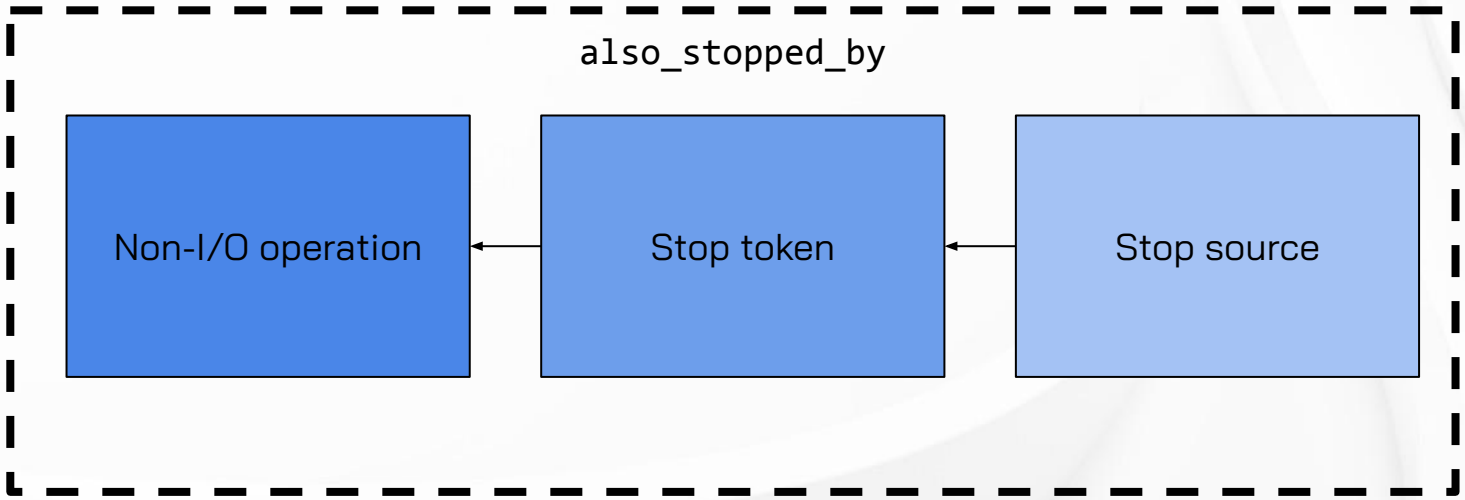


Conditional Stop Token

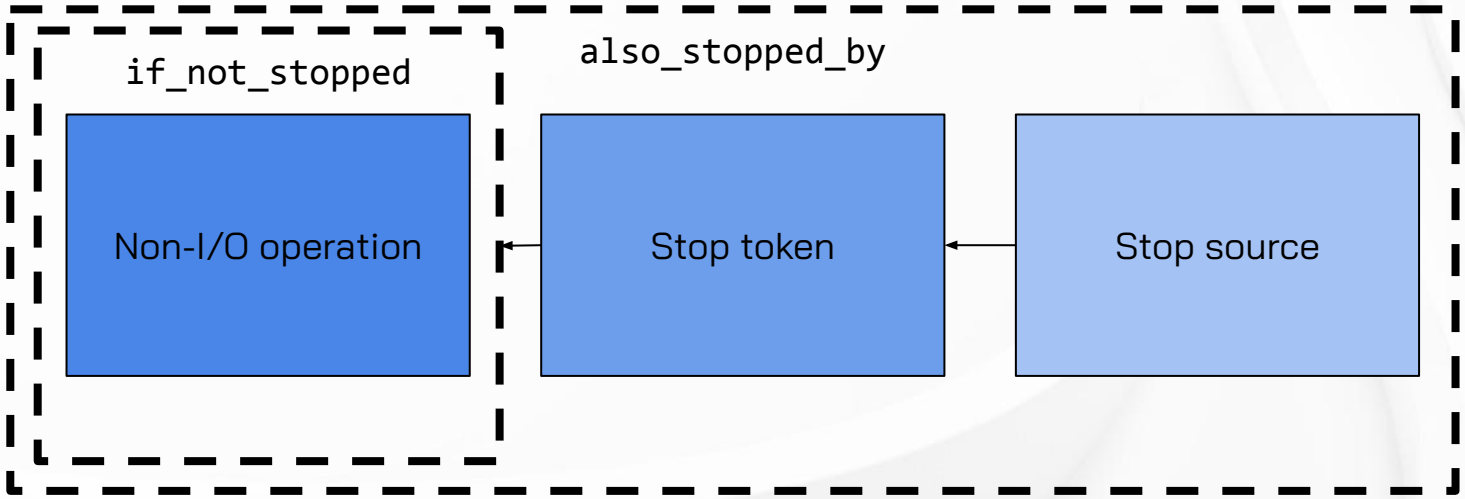


Stops Aren’t Mandatory

Mandatory Stop



Mandatory Stop



Running a Request

This is the function signature.

```
template<
  typename Query,
  typename Scheduler,
  typename Timer,
  typename Duration,
  typename Writable,
  typename Token>
auto request(
  Query& q,
  Scheduler sch,
  Timer& t,
  Duration poll_interval,
  Writable& writable,
  mutex& write_lock,
  const Token& cancel,
  std::span<const std::byte> request);
```

LSEG

Scheduler?

What's this?

```
template<
  typename Query,
  typename Scheduler,
  typename Timer,
  typename Duration,
  typename Writable,
  typename Token>
auto request(
  Query& q,
  Scheduler sch,
  Timer& t,
  Duration poll_interval,
  Writable& writable,
  mutex& write_lock,
  const Token& cancel,
  std::span<const std::byte> request);
```

LSEG

Running Arbitrary Work

```
template<class Sch>
  concept scheduler =
    derived_from<
      typename remove_cvref_t<Sch>::scheduler_concept,
      scheduler_t> &&
    queryable<Sch> &&
    requires(Sch&& sch) {
      { schedule(std::forward<Sch>(sch)) } -> sender;
      { auto(get_completion_scheduler<set_value_t>(
          get_env(schedule(std::forward<Sch>(sch))))) }
            -> same_as<remove_cvref_t<Sch>>;
    } &&
    equality_comparable<remove_cvref_t<Sch>> &&
    copy_constructible<remove_cvref_t<Sch>>;
```

Final Interface

`epoll` extended to support providing a scheduler.

```
struct epoll {
  //  ...
  struct scheduler {
    using scheduler_concept = std::execution::scheduler_t;
    std::execution::sender auto schedule() const noexcept;
    bool operator==(const scheduler&) const noexcept;
  };
  scheduler get_scheduler() noexcept;
};
```

LSEG

Timer?

What's this?

```
template<
  typename Query,
  typename Scheduler,
  typename Timer,
  typename Duration,
  typename Writable,
  typename Token>
auto request(
  Query& q,
  Scheduler sch,
  Timer& t,
  Duration poll_interval,
  Writable& writable,
  mutex& write_lock,
  const Token& cancel,
  std::span<const std::byte> request);
```

LSEG

NAME top

timerfd_create, timerfd_settime, timerfd_gettime - timers that notify via file descriptors

LIBRARY top

Standard C library (*libc*, *-lc*)

SYNOPSIS top

```
#include <sys/timerfd.h>

int timerfd_create(int clockid, int flags);

int timerfd_settime(int fd, int flags,
```

Timer

Simple wrapper around a timerfd.

```
struct timer {
  explicit timer(epoll& ep);
  template<typename Rep, typename Period>
  std::execution::sender auto wait_for(
    std::chrono::duration<Rep, Period> d) noexcept;
};
```

LSEG

mutex

What's this?

```
template<
  typename Query,
  typename Scheduler,
  typename Timer,
  typename Duration,
  typename Writable,
  typename Token>
auto request(
  Query& q,
  Scheduler sch,
  Timer& t,
  Duration poll_interval,
  Writable& writable,
  mutex& write_lock,
  const Token& cancel,
  std::span<const std::byte> request);
```

LSEG

DataConn

TCP protocol without login or encryption
    Intended solely for use on private network

Client (front-end) sends request

Server (back-end) accepts or rejects

Response streamed until final message or until client requests cancel
    Server acknowledges cancel

After response connection is ready for another request

Server sends heartbeats during inactivity

Mutex

`with` returns a sender which runs the wrapped operation while holding the lock.

```
struct mutex {
  template<std::execution::sender Sender>
  std::execution::sender auto with(Sender&& sender);
};
```

Running a Request

Compressing the function signature down to give us more room to work with.

```
auto request(Query& q, Scheduler sch, Timer& t,
  Duration poll_interval, Writable& writable, mutex& write_lock,
  const Token& cancel, std::span<const std::byte> request)
{
  /*  ...
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   */
}
```

LSEG

Running a Request

Writes a buffer of bytes while holding the lock (to prevent torn heartbeat transmission).

```
auto request(Query& q, Scheduler sch, Timer& t,
  Duration poll_interval, Writable& writable, mutex& write_lock,
  const Token& cancel, std::span<const std::byte> request)
{
  const auto write = [&writable, &write_lock](
    const data_conn::message_type type,
    const std::span<const std::byte> payload)
  {
    return write_lock.with(
      data_conn::write(writable, type, payload));
  };
  /*  ...
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   */
}
```

LSEG

Running a Request

Writes the next payload from the query and flips the buffers for concurrent generation.

```
auto request(Query& q, Scheduler sch, Timer& t,
  Duration poll_interval, Writable& writable, mutex& write_lock,
  const Token& cancel, std::span<const std::byte> request)
{
  //  ...
  const auto write_next = [&q, write](
    const data_conn::message_type type)
  {
    const auto s = q.payload();
    q.toggle();
    return write(type, s);
  };
  /*  ...
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   */
}
```

Running a Request

This makes some code later simpler.

```
auto request(Query& q, Scheduler sch, Timer& t,
  Duration poll_interval, Writable& writable, mutex& write_lock,
  const Token& cancel, std::span<const std::byte> request)
{
  //  ...
  const auto end = [&q]() noexcept { q.end(); };
  /*  ...
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   */
}
```

LSEG

Running a Request

Allows a different action to be chosen if cancel is requested (as opposed to stop).

```
auto request(Query& q, Scheduler sch, Timer& t,
  Duration poll_interval, Writable& writable, mutex& write_lock,
  const Token& cancel, std::span<const std::byte> request)
{
  //  ...
  const auto if_canceled = [cancel](auto&& canceled) {
    auto stopped = std::execution::just_stopped();
    using type = ::exec::variant_sender<
      decltype(stopped),
      std::remove_cvref_t<decltype(canceled)>>;
    if (cancel.stop_requested()) {
      return type(std::forward<decltype(canceled)>(canceled));
    }
    return type(std::move(stopped));
  };
  /*  ...
   *
   *
   *
   *
   *
   *
   *
   *
   */
}
```

LSEG

Running a Request

`if_not_stopped` adds eager stop checking.

`also_stopped_by` adds the stop requests from `cancel` to those from the stop token in the eventually-connected receiver's environment.

```
auto request(Query& q, Scheduler sch, Timer& t,
  Duration poll_interval, Writable& writable, mutex& write_lock,
  const Token& cancel, std::span<const std::byte> request)
{
  //  ...
  const auto enable_cancel = [cancel](auto&& sender) {
    return
      if_not_stopped(
        std::forward<decltype(sender)>(sender)) |
      also_stopped_by(cancel);
  };
  /*  ...
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   */
}
```

LSEG

Running a Request

Generates the next payload with cancellation support.

```
auto request(Query& q, Scheduler sch, Timer& t,
  Duration poll_interval, Writable& writable, mutex& write_lock,
  const Token& cancel, std::span<const std::byte> request)
{
  //  ...
  const auto run = enable_cancel(run(q, sch, t, poll_interval));
  /*  ...
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   */
}
```

LSEG

Running a Request

A cancellation-aware sender which performs initialization.

```
auto request(Query& q, Scheduler sch, Timer& t,
  Duration poll_interval, Writable& writable, mutex& write_lock,
  const Token& cancel, std::span<const std::byte> request)
{
  //  ...
  return
    enable_cancel(init(q, sch, t, poll_interval, request)))









}
```

LSEG

Running a Request

Rejection results in a write.

Acceptance results in more complicated actions which will be presented later.

```
auto request(Query& q, Scheduler sch, Timer& t,
  Duration poll_interval, Writable& writable, mutex& write_lock,
  const Token& cancel, std::span<const std::byte> request)
{
  //  ...
  return
    enable_cancel(init(q, sch, t, poll_interval, request))) |
    std::execution::let_value(
      overload(
        [=]<typename BufferSequence>(
          const async_control_reject<BufferSequence>& reject)
        {
          return write(
            message_type::reject_request, reject.payload);
        },
        [=]<typename BufferSequence>(
          const async_control_accept<BufferSequence>& accept)
        { /* ... */ }))




}
```

Running a Request

We want to write a cancel acknowledgment.

```
auto request(Query& q, Scheduler sch, Timer& t,
  Duration poll_interval, Writable& writable, mutex& write_lock,
  const Token& cancel, std::span<const std::byte> request)
{
  //  ...
  return
    enable_cancel(init(q, sch, t, poll_interval, request))) |
    std::execution::let_value(
      overload(
        [=]<typename BufferSequence>(
          const async_control_reject<BufferSequence>& reject)
        {
          return write(
            message_type::reject_request, reject.payload);
        },
        [=]<typename BufferSequence>(
          const async_control_accept<BufferSequence>& accept)
        { /* ... */ })) |
    std::execution::let_stopped([=]() {
      return

          write(message_type::acknowledge_cancel, {}) |
          std::execution::then(end)  ;
    });
}
```

LSEG

Running a Request

But we only do that if we've been canceled.

```
auto request(Query& q, Scheduler sch, Timer& t,
  Duration poll_interval, Writable& writable, mutex& write_lock,
  const Token& cancel, std::span<const std::byte> request)
{
  //  ...
  return
    enable_cancel(init(q, sch, t, poll_interval, request))) |
    std::execution::let_value(
      overload(
        [=]<typename BufferSequence>(
          const async_control_reject<BufferSequence>& reject)
        {
          return write(
            message_type::reject_request, reject.payload);
        },
        [=]<typename BufferSequence>(
          const async_control_accept<BufferSequence>& accept)
        { /* ... */ })) |
    std::execution::let_stopped([=]() {
      return
        if_canceled(
          write(message_type::acknowledge_cancel, {}) |
          std::execution::then(end)) ;
    });
}
```

LSEG

Running a Request

Which we only want to do if we weren't stopped.

```
auto request(Query& q, Scheduler sch, Timer& t,
  Duration poll_interval, Writable& writable, mutex& write_lock,
  const Token& cancel, std::span<const std::byte> request)
{
  //  ...
  return
    enable_cancel(init(q, sch, t, poll_interval, request))) |
    std::execution::let_value(
      overload(
        [=]<typename BufferSequence>(
          const async_control_reject<BufferSequence>& reject)
        {
          return write(
            message_type::reject_request, reject.payload);
        },
        [=]<typename BufferSequence>(
          const async_control_accept<BufferSequence>& accept)
        { /* ... */ })) |
    std::execution::let_stopped([=]() {
      return if_not_stopped(
        if_canceled(
          write(message_type::acknowledge_cancel, {}) |
          std::execution::then(end)));
    });
}
```

LSEG

Running a Request

First write the acknowledgement.

```
[=]<typename BufferSequence>(
  const async_control_accept<BufferSequence>& accept)
{
  return ::exec::sequence(
    write(
      message_type::acknowledge_request,
      accept.payload),
                                    );
}
```

LSEG

Running a Request

Loop for each payload.

```
[=]<typename BufferSequence>(
  const async_control_accept<BufferSequence>& accept)
{
  return ::exec::sequence(
    write(
      message_type::acknowledge_request,
      accept.payload),
    ::exec::repeat_effect_until(




        )
                                  );
}
```

LSEG

Running a Request

Allow the query to generate the next payload.

```
[=]<typename BufferSequence>(
  const async_control_accept<BufferSequence>& accept)
{
  return ::exec::sequence(
    write(
      message_type::acknowledge_request,
      accept.payload),
    ::exec::repeat_effect_until(
      run |
      std::execution::let_value([=](const message_type type) {




      }))
                                  );
}
```

LSEG

Running a Request

Once the payload is generated, send it.

```
[=]<typename BufferSequence>(
  const async_control_accept<BufferSequence>& accept)
{
  return ::exec::sequence(
    write(
      message_type::acknowledge_request,
      accept.payload),
    ::exec::repeat_effect_until(
      run |
      std::execution::let_value([=](const message_type type) {
        return ::exec::sequence(
          write_next(type),

                                                    );
      }))
                                );
}
```

LSEG

Running a Request

Termination condition.

```
[=]<typename BufferSequence>(
  const async_control_accept<BufferSequence>& accept)
{
  return ::exec::sequence(
    write(
      message_type::acknowledge_request,
      accept.payload),
    ::exec::repeat_effect_until(
      run |
      std::execution::let_value([=](const message_type type) {
        return ::exec::sequence(
          write_next(type),
          std::execution::just(
            type != message_type::intermediate_chunk));
      }))
                                );
}
```

LSEG

Running a Request

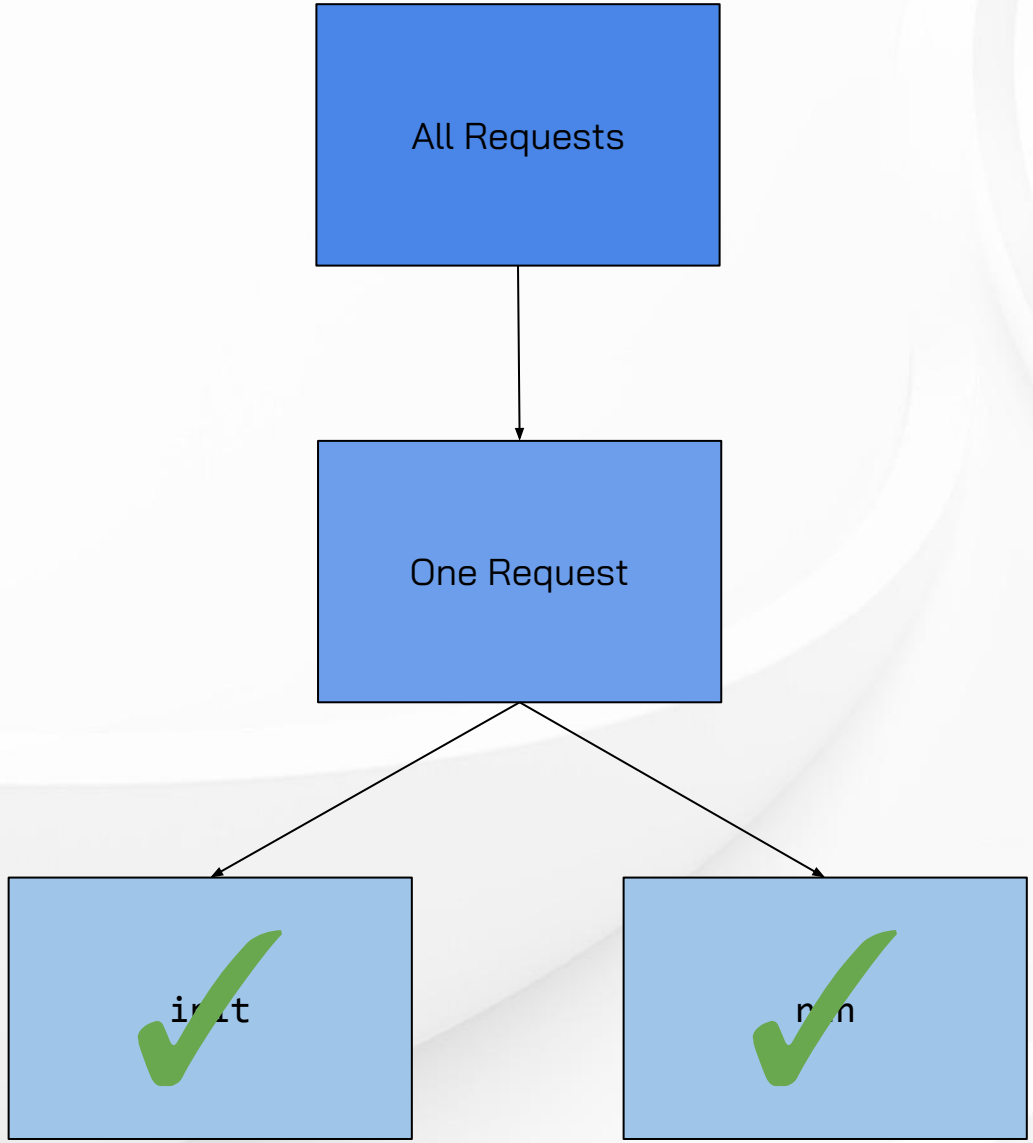
Finalize the query to finish things off.

```
[=]<typename BufferSequence>(
  const async_control_accept<BufferSequence>& accept)
{
  return ::exec::sequence(
    write(
      message_type::acknowledge_request,
      accept.payload),
    ::exec::repeat_effect_until(
      run |
      std::execution::let_value([=](const message_type type) {
        return ::exec::sequence(
          write_next(type),
          std::execution::just(
            type != message_type::intermediate_chunk));
      })) |
    std::execution::then(end));
}
```

LSEG

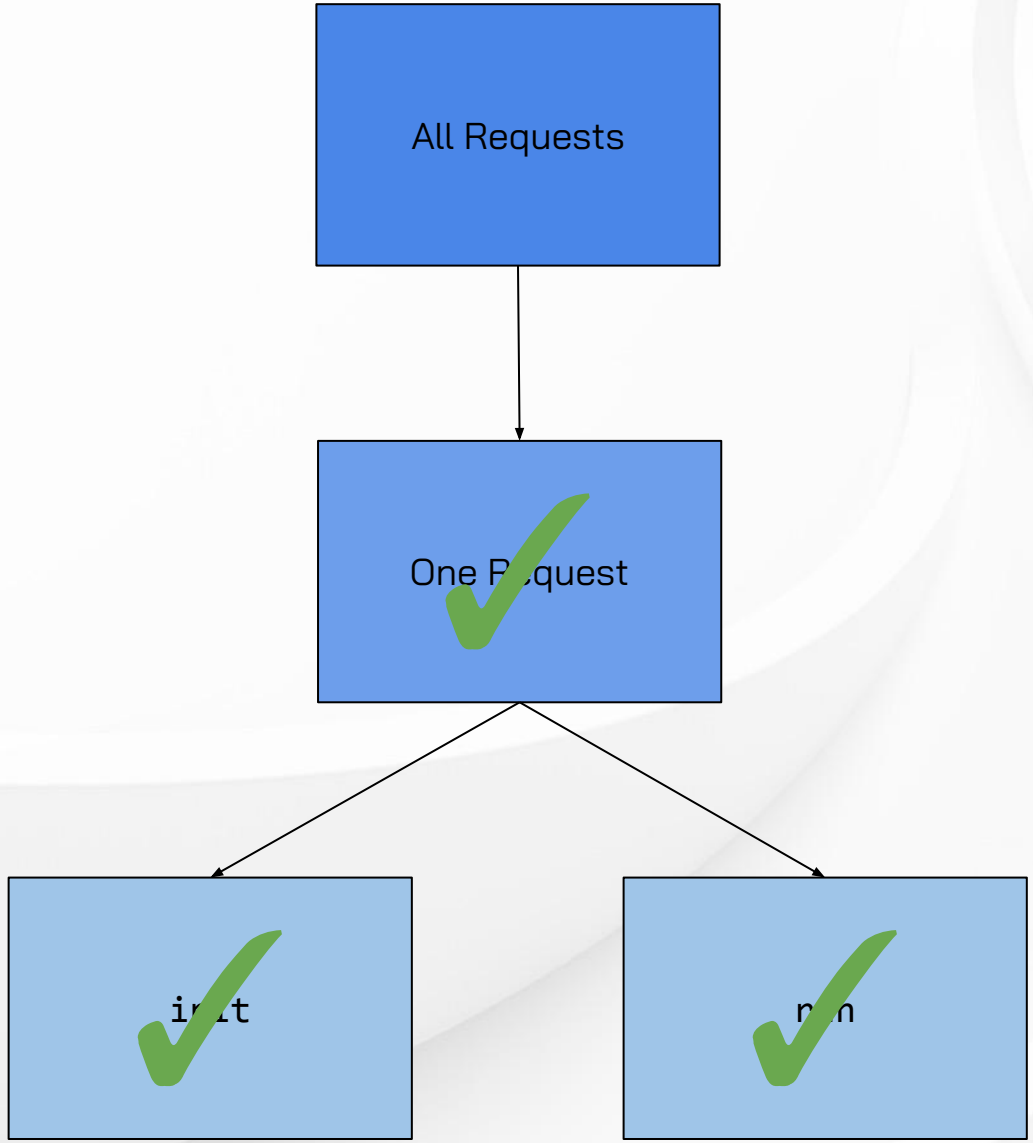
What Next?

Building on Top of Our Primitives



What Next?

Building on Top of Our Primitives



LSEG

Controlling a Connection

Factory is an invocable which provides the sender to process each request.

```
template<typename Readable, typename Factory>
auto control(Readable& readable, Factory f);
```

Controlling a Connection

Everything that is shared between the two concurrent parts of the `control` operation.

```
namespace detail::control {

struct state {
  queue<std::vector<std::byte>> requests;
  std::list<std::inplace_stop_source> cancels;
  std::vector<std::byte> buffer;
};

}
```

LSEG

queue?

What's this?

```
namespace detail::control {

struct state {
  queue<std::vector<std::byte>> requests;
  std::list<std::inplace_stop_source> cancels;
  std::vector<std::byte> buffer;
};

}
```

LSEG

queue

Consumer waits asynchronously when queue is empty.

```
template<typename T>
struct queue {
  template<typename... Args>
  void push(Args&&... args);
  std::execution::sender auto pop();
};
```

Controlling a Connection

Read part of the operation simply reads messages in a loop.

```
namespace detail::control {

template<typename Readable>
auto read(Readable& readable, state& state) {









    }));
}

}
```

LSEG

Controlling a Connection

We want to repeatedly perform an operation until there's an error or we're requested to stop.

```
namespace detail::control {

template<typename Readable>
auto read(Readable& readable, state& state) {
  return repeat(









      );
}

}
```

LSEG

repeat?

```
namespace detail::control {

template<typename Readable>
auto read(Readable& readable, state& state) {
  return repeat(
    data_conn::read(readable, state.buffer) |
    std::execution::then([&](const message_type type) {









    }));
}

}
```

LSEG

repeat?

Requests are submitted via the queue.

```
namespace detail::control {

template<typename Readable>
auto read(Readable& readable, state& state) {
  return repeat(
    data_conn::read(readable, state.buffer) |
    std::execution::then([&](const message_type type) {
      if (type == message_type::request) {
        state.cancels.emplace_back();
        state.requests.push(
          std::exchange(state.buffer, std::vector<std::byte>{}));
        return;
      }









    }));
}

}
```

LSEG

repeat?

If cancellation is requested “pop:” Find the latest uncanceled request and cancel it.

```
namespace detail::control {

template<typename Readable>
auto read(Readable& readable, state& state) {
  return repeat(
    data_conn::read(readable, state.buffer) |
    std::execution::then([&](const message_type type) {
      if (type == message_type::request) {
        state.cancels.emplace_back();
        state.requests.push(
          std::exchange(state.buffer, std::vector<std::byte>{}));
        return;
      }
      if (const auto iter = std::find_if(
        state.cancels.rbegin(),
        state.cancels.rend(),
        [](const auto& source) noexcept {
          return !source.get_token().stop_requested();
        });
        iter != state.cancels.rend())
      { iter->request_stop(); }
    }));
}

}
```

LSEG

Controlling a Connection

This concurrent part will pop the queue and process requests.

```
namespace detail::control {

template<factory Factory>
auto run(Factory f, state& state) {









}

}
```

Controlling a Connection

```
namespace detail::control {

template<factory Factory>
auto run(Factory f, state& state) {
  return repeat(
    state.requests.pop() |
    std::execution::let_value([f = std::move(f)](
      std::vector<std::byte>& v) mutable
    {









    }));
}

}
```

LSEG

Controlling a Connection

Obtain the sender which processes the request from the factory invocable.

```
namespace detail::control {

template<factory Factory>
auto run(Factory f, state& state) {
  return repeat(
    state.requests.pop() |
    std::execution::let_value([f = std::move(f)](
      std::vector<std::byte>& v) mutable
    {
      return
        std::invoke(
          f,
          state.cancels.front().get_token(),
          std::move(v))



    }));
}

}
```

LSEG

Controlling a Connection

After processing clean up the cancellation stop token.

```
namespace detail::control {

template<factory Factory>
auto run(Factory f, state& state) {
  return repeat(
    state.requests.pop() |
    std::execution::let_value([f = std::move(f)](
      std::vector<std::byte>& v) mutable
    {
      return
        std::invoke(
          f,
          state.cancels.front().get_token(),
          std::move(v)) |
        std::execution::then([&]() noexcept {
          state.cancels.pop_front();
        });
    }));
}

}
```

Controlling a Connection

Bringing both pieces together.

```
template<typename Readable, typename Factory>
auto control(Readable& readable, Factory f) {
  return with<detail::control::state>([
    &readable,
    f = std::move(f)](detail::control::state& state) mutable
  {



  });
}
```

LSEG

with?

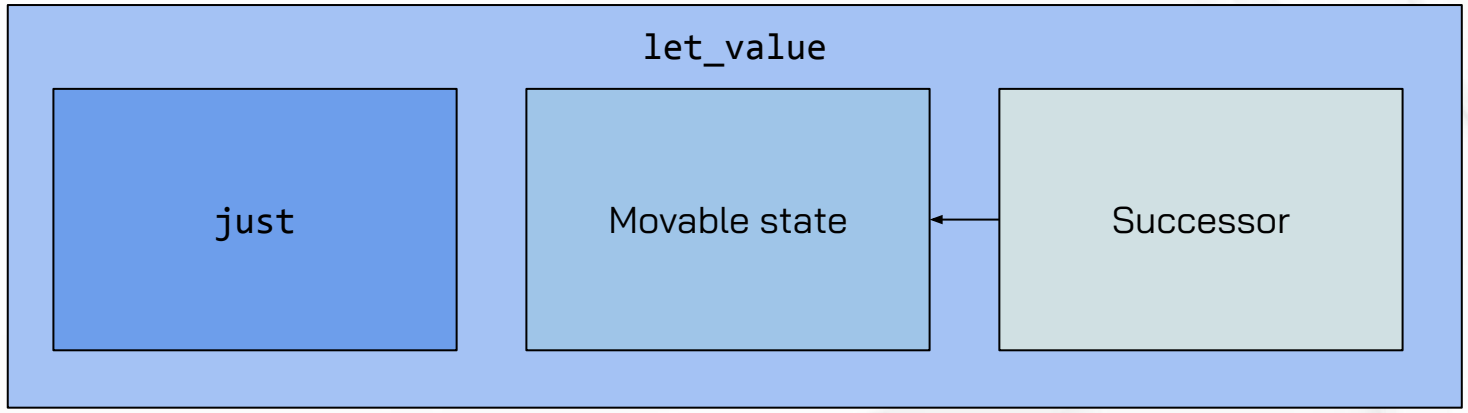
Canned version of the `std::execution::just` piped into `std::execution::let_value` idiom which supports immovable types.

```
template<typename Readable, typename Factory>
auto control(Readable& readable, Factory f) {
  return with<detail::control::state>(
    &readable,
    f = std::move(f)](detail::control::state& state) mutable
  {



  });
}
```

LSEG

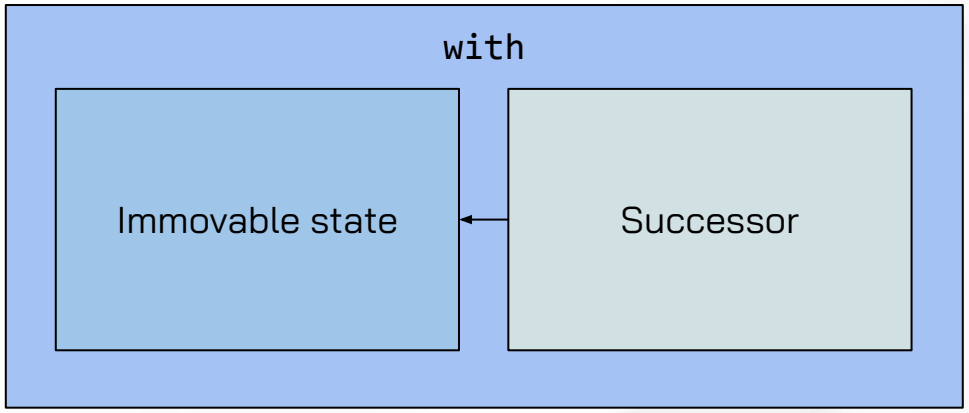
Operation State Nesting

`std::execution::let_value`



Operation State Nesting

`with`



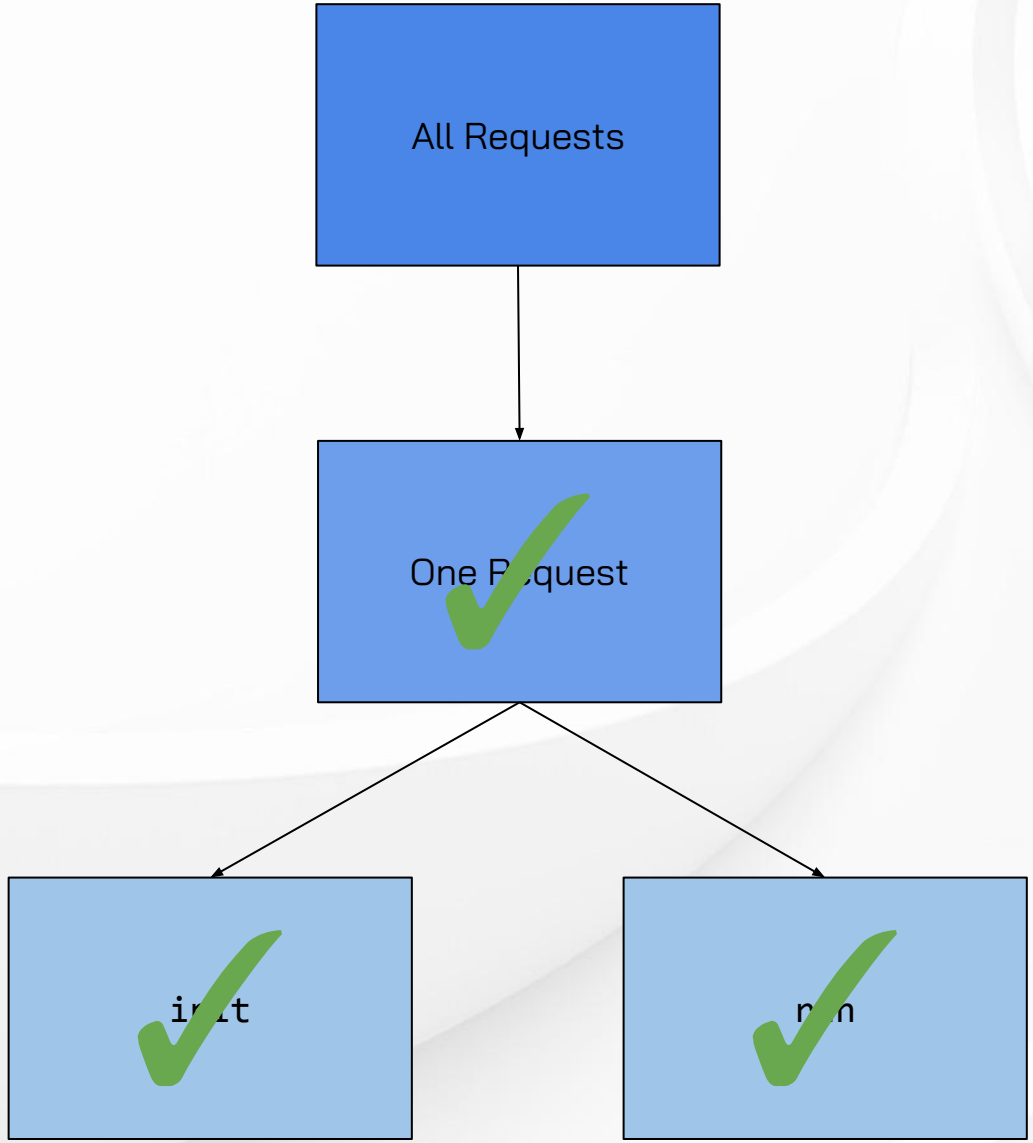
Controlling a Connection

Using `std::execution::when_all` we start both concurrent parts.

```
template<typename Readable, typename Factory>
auto control(Readable& readable, Factory f) {
  return with<detail::control::state>([
    &readable,
    f = std::move(f)](detail::control::state& state) mutable
  {
    return std::execution::when_all(
      detail::control::read(readable, state),
      detail::control::run(std::move(f), state));
  });
}
```

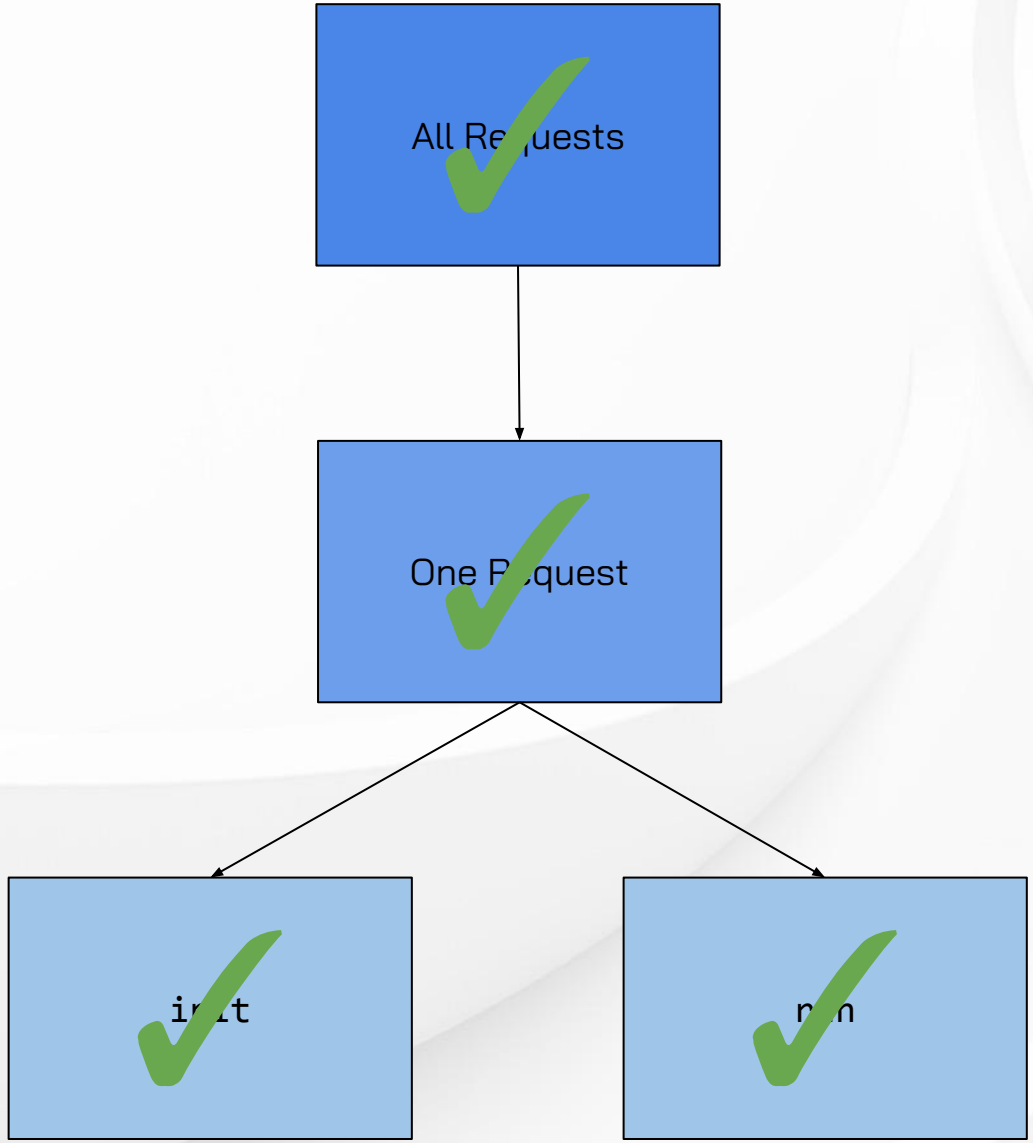
What Next?

Building on Top of Our Primitives



What Next?

Building on Top of Our Primitives

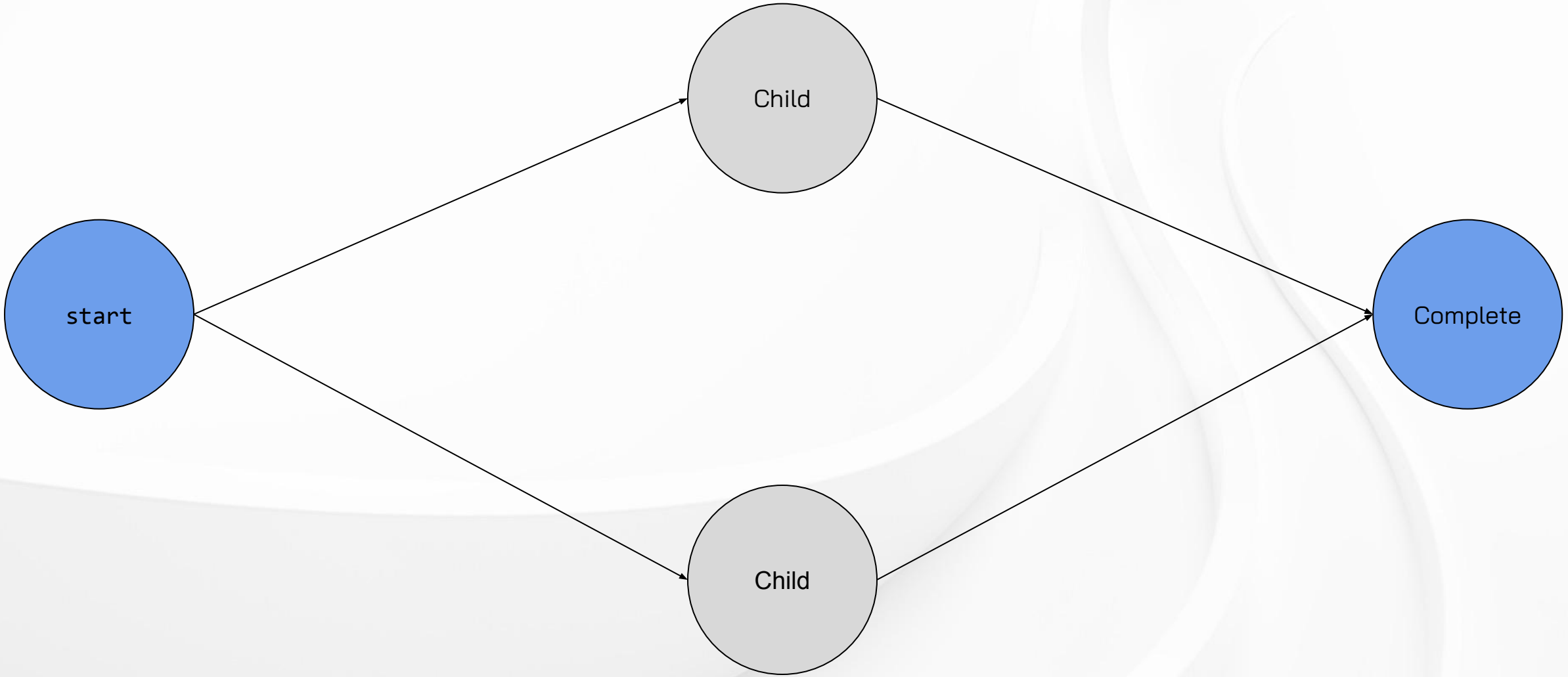


LSEG


Dynamic Fan Out

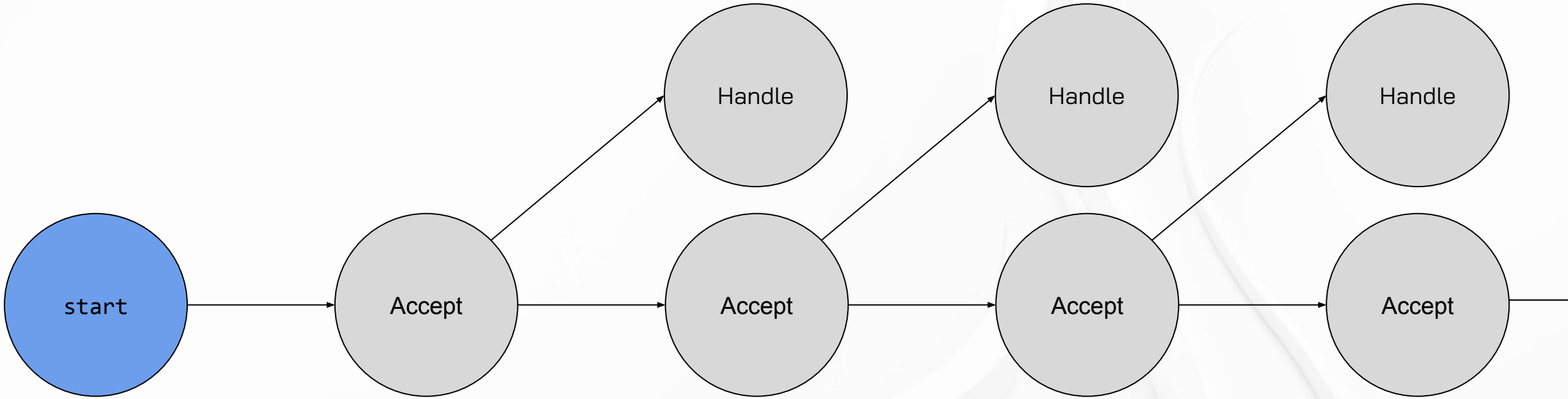
when_all

Fan Out, Fan In



Accepting

Fan Out, Repeat



branch

Unary sender factory, but what kind of sender does it accept?

```
template<std::execution::sender Sender>
std::execution::sender auto branch(Sender&& sender);
```


Higher order sender

```
int main(int argc, const char** argv);
```

main

Simply setting the stage, parsing the command line arguments and declaring a per connection state.

Note that `query` is the type of some query implementation.

```
program_options_parser pop;
//  ...
struct state {
  explicit state(
    epoll& ctx,
    ::exec::safe_file_descriptor fd)
    : fd(ctx, std::move(fd)),
      timer(ctx),
      heartbeat_timer(ctx)
  {}
  file_descriptor fd;
  impl::timer timer;
  impl::timer heartbeat_timer;
  impl::query query;
  impl::mutex mutex;
};
```

main

Generates a sender to handle each newly-accepted connection.

```
const auto connection = [&](
  epoll& ctx,
  ::exec::safe_file_descriptor accepted)
{
  return with<state>(
    [&](state& s) {
      return
        std::execution::when_all(/* ... */) |
        std::execution::upon_error([](std::exception_ptr ex) {
          //  ...
        });
    },
    std::ref(ctx),
    std::move(accepted));
};
```

LSEG

main

Handling each connection consists of two concurrent operations.

```
std::execution::when_all(

                                                    )
```

main

The first, `control`, reads requests and processes them.

But what action is performed for each request?

```
std::execution::when_all(
  control(
    s.fd,
    [&](auto&& cancel, auto&& buffer) {

    }),

                                      )
```

LSEG

main

We wrote an operation for that earlier.

```
std::execution::when_all(
  control(
    s.fd,
    [&](auto&& cancel, auto&& buffer) {
      return request(
        s.query,
        ctx.get_scheduler(),
        s.timer,
        pop.query_driver_settings().poll_interval,
        s.fd,
        s.mutex,
        std::forward<decltype(cancel)>(cancel),
        std::forward<decltype(buffer)>(buffer));
    }),
                                                )
```

LSEG

main

Second concurrent operation is sending periodic heartbeats.

```
std::execution::when_all(
  control(
    s.fd,
    [&](auto&& cancel, auto&& buffer) {
      return request(
        s.query,
        ctx.get_scheduler(),
        s.timer,
        pop.query_driver_settings().poll_interval,
        s.fd,
        s.mutex,
        std::forward<decltype(cancel)>(cancel),
        std::forward<decltype(buffer)>(buffer));
    }),
  repeat(
```

```
                                                  ))
```

LSEG

main

First we wait.

```
std::execution::when_all(
  control(
    s.fd,
    [&](auto&& cancel, auto&& buffer) {
      return request(
        s.query,
        ctx.get_scheduler(),
        s.timer,
        pop.query_driver_settings().poll_interval,
        s.fd,
        s.mutex,
        std::forward<decltype(cancel)>(cancel),
        std::forward<decltype(buffer)>(buffer));
    }),
  repeat(
    ::exec::sequence(
      s.heartbeat_timer.wait_for(pop.heartbeat_interval()),
                                                    )))
```

LSEG

main

Then we write.

```
std::execution::when_all(
  control(
    s.fd,
    [&](auto&& cancel, auto&& buffer) {
      return request(
        s.query,
        ctx.get_scheduler(),
        s.timer,
        pop.query_driver_settings().poll_interval,
        s.fd,
        s.mutex,
        std::forward<decltype(cancel)>(cancel),
        std::forward<decltype(buffer)>(buffer));
    }),
  repeat(
    ::exec::sequence(
      s.heartbeat_timer.wait_for(pop.heartbeat_interval()),
      s.mutex.with(
        data_conn::write(
          s.fd,
          message_type::heartbeat)))))
```

LSEG

main

Creates an accepting socket and repeatedly accepts thereupon.

```
const auto accept = [&](epoll& ctx) {









};
```

main

Create a `file_descriptor` wrapping an accepting socket for the endpoint.

```
const auto accept = [&](epoll& ctx) {
  return with<file_descriptor>(
    [&](file_descriptor& fd) {




    },
    std::ref(ctx),
    make_listening_socket(pop.endpoint()));
};
```

LSEG

main

`branch` will be used to repeatedly accept and spawn a new operation for each accepted connection.

```
const auto accept = [&](epoll& ctx) {
  return with<file_descriptor>(
    [&](file_descriptor& fd) {
      return branch(




          );
    },
    std::ref(ctx),
    make_listening_socket(pop.endpoint()));
};
```

LSEG

main

Accepting is the main line off which operations branch.

```
const auto accept = [&](epoll& ctx) {
  return with<file_descriptor>(
    [&](file_descriptor& fd) {
      return branch(
        fd.accept()




          );
    },
    std::ref(ctx),
    make_listening_socket(pop.endpoint()));
};
```

LSEG

main

`branch` accepts a higher order sender, so we coalesce the value “returned” by accepting into a sender which handles the newly-accepted connection.

```
const auto accept = [&](epoll& ctx) {
  return with<file_descriptor>(
    [&](file_descriptor& fd) {
      return branch(
        fd.accept() |
        std::execution::then([&](
          ::exec::safe_file_descriptor accepted,
          const ::sockaddr_storage&)
        {
          return connection(ctx, std::move(accepted));
        }));
    },
    std::ref(ctx),
    make_listening_socket(pop.endpoint()));
};
```

LSEG

main

`sync_wait_until_stopped` is a custom version of `std::execution::sync_wait` which handles senders with no value completions (we only send stopped and error).

```
sync_wait_until_stopped(
                );
```

main

`run` generates an `epoll`, injects it into the provided callable, and ensures it's run to completion.

```
sync_wait_until_stopped(
  run(accept));
```

Does it work?

```
2024-10-27T03:06:56.309741016Z
  Sent 8 bytes
  Request (R)
  foo
2024-10-27T03:06:56.309899670Z
  Received 19 bytes
  Acknowledge Request (A)
  Query accepted
2024-10-27T03:06:56.310023806Z
  Received 11 bytes
  Intermediate Chunk (I)
  Hello
2024-10-27T03:06:59.309269032Z
  Received 5 bytes
  Heartbeat (H)
2024-10-27T03:07:01.317230230Z
  Received 11 bytes
  Final Chunk (F)
  world!
```

```
2024-10-27T03:06:56.309741016Z
  Sent 8 bytes
  Request (R)
  foo
2024-10-27T03:06:56.309899670Z
  Received 19 bytes
  Acknowledge Request (A)
  Query accepted
2024-10-27T03:06:56.310023806Z
  Received 11 bytes
  Intermediate Chunk (I)
  Hello
2024-10-27T03:06:59.309269032Z
  Received 5 bytes
  Heartbeat (H)
2024-10-27T03:07:01.317230230Z
  Received 11 bytes
  Final Chunk (F)
  world!
```

```
2024-10-27T03:08:01.667723819Z
  Sent 8 bytes
  Request (R)
  foo
2024-10-27T03:08:01.667942863Z
  Received 19 bytes
  Acknowledge Request (A)
  Query accepted
2024-10-27T03:08:01.668042233Z
  Received 11 bytes
  Intermediate Chunk (I)
  Hello
2024-10-27T03:08:03.107771075Z
  Received 5 bytes
  Heartbeat (H)
2024-10-27T03:08:03.885230568Z
  Sent 5 bytes
  Cancel (C)
2024-10-27T03:08:03.885414738Z
  Received 5 bytes
  Acknowledge Cancel (K)
```

Thank you

Robert Leahy
Lead Software Engineer
rleahy@rleahy.ca

LSEG

Questions?

LSEG