C++ now

2025

Extending std::execution

Implementing Custom Algorithms with Senders & Receivers

Robert Leahy

Let’s invoke something

std::invoke, std::invoke_r

Defined in header <functional>

```
template< class F, class... Args >
std::invoke_result_t<F, Args...>
    invoke( F&& f, Args&&... args ) noexcept(/* see below */);
```
(1) (since C++17) (constexpr since C++20)

```
template< class R, class F, class... Args >
constexpr R
    invoke_r( F&& f, Args&&... args ) noexcept(/* see below */);
```
(2) (since C++23)

1) Invoke the *Callable* object `f` with the parameters `args` as by `INVOKE(std::forward<F>(f), std::forward<Args>(args)...)`. This overload participates in overload resolution only if `std::is_invocable_v<F, Args...>` is `true`.

2) Invoke the *Callable* object `f` with the parameters `args` as by `INVOKE<R>(std::forward<F>(f), std::forward<Args>(args)...)`. This overload participates in overload resolution only if `std::is_invocable_r_v<R, F, Args...>` is `true`.

Parameters

f - *Callable* object to be invoked

args - arguments to pass to `f`

Return value

1) The value returned by `f`.

2) The value returned by `f`, implicitly converted to R, if R is not (possibly cv-qualified) `void`. None otherwise.

invoke

Does something by invoking a nullary invocable.

```
template<typename T>
std::invoke_result_t<T> invoke(T&& t) noexcept(
  std::is_nothrow_invocable_v<T>)
{
  return std::invoke(std::forward<T>(t));
}
```

LSEG

Let’s invoke several things

invoke_all

Does several things by invoking several invocables

```
template<typename T>
  requires std::is_same_v<std::invoke_result_t<T>, void>
std::tuple<> invoke_impl(T&& t) noexcept(/* ... */) {
    std::invoke(std::forward<T>(t));
    return {};
}

template<typename T>
  requires std::is_move_constructible_v<
    std::invoke_result_t<T>>
auto invoke_impl(T&& t) noexcept(/* ... */) {
  return std::tuple<std::invoke_result_t<T>>(
    std::invoke(std::forward<T>(t)));
}

template<typename... Ts>
  requires requires {
    (::invoke_impl(std::declval<Ts>()), ...);
  }
auto invoke_all(Ts&&... ts) noexcept(/* ... */) {
  return std::tuple_cat(
    ::invoke_impl(std::forward<Ts>(ts))...);
}
```

LSEG

Problem

What's the behavior of this program?

```
const auto a = []() {
  std::this_thread::sleep_for(std::chrono::seconds(1));
  std::cout << "a" << std::endl;
};
const auto b = []() {
  std::this_thread::sleep_for(std::chrono::seconds(2));
  std::cout << "b" << std::endl;
};
invoke_all(a, b);
```

LSEG

a

b

Run time is three seconds

LSEG

std::execution::when_all

Defined in header <execution>

```
execution::sender auto when_all( execution::sender auto... inputs );
```
(since C++26)

Parameters

inputs - senders upon which the completion of `when_all` is blocked. Can only include senders that can complete with a single set of values.

Return value

Returns a sender that completes once all of the input senders have completed. The values sent by this sender are the values sent by each of the input senders, in order of the arguments passed to `when_all`.

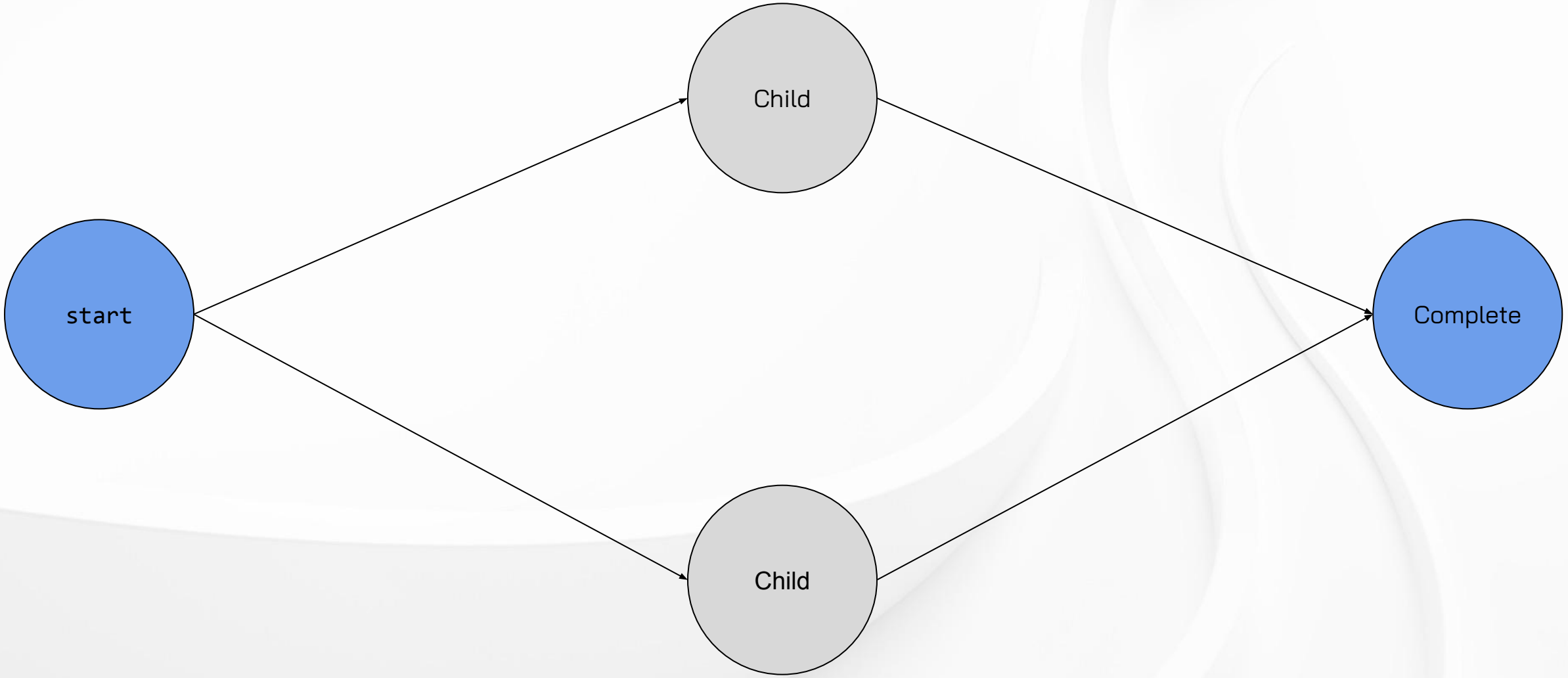
Notes

- The sender returned by `when_all` completes inline on the execution resource on which the last input sender completes, unless `stop` is requested before `when_all` is started, in which case it completes inline within the call to start.

Example

This section is incomplete
Reason: no example

See also

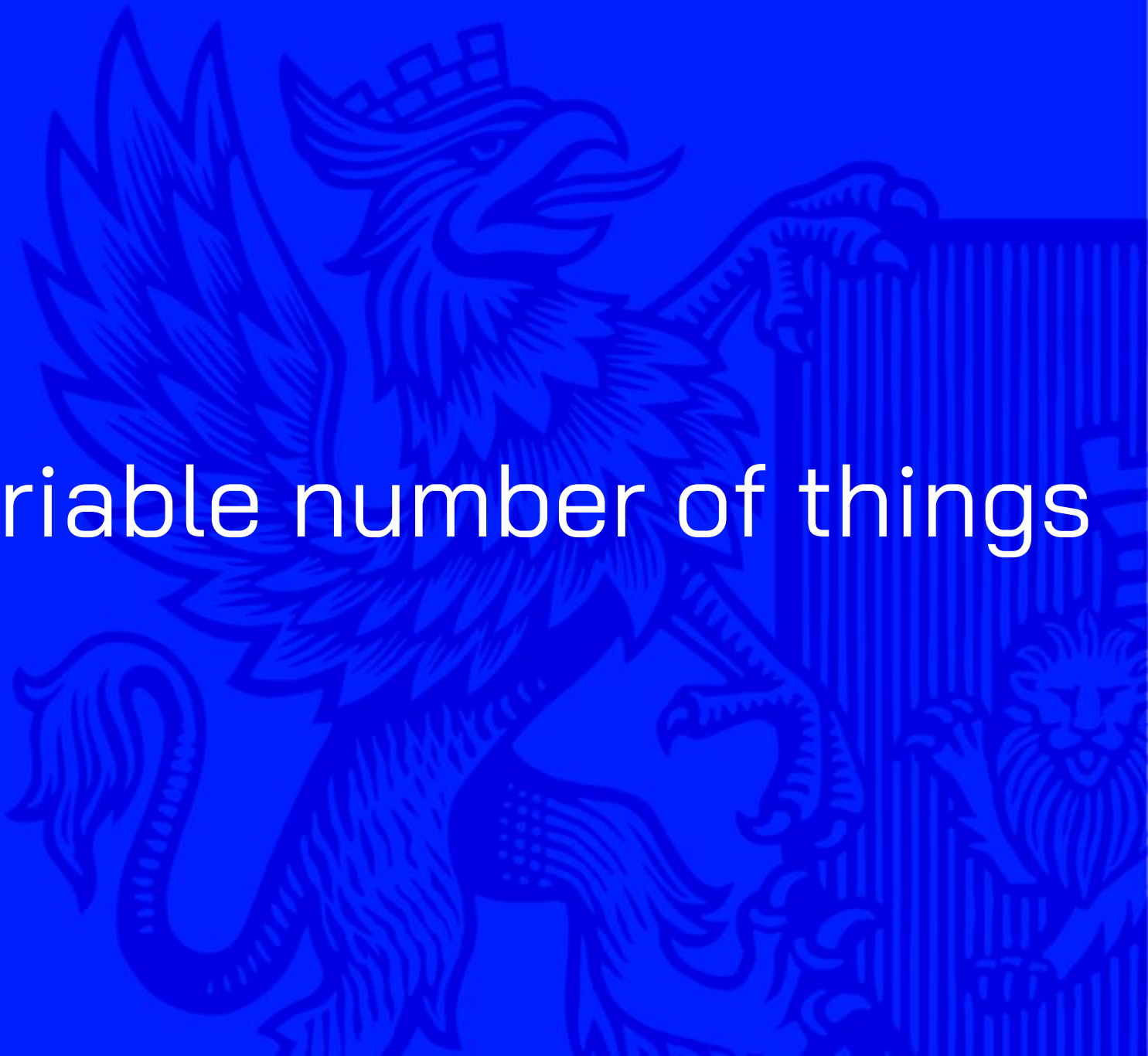

start
Child
Child
Complete

Solution

Runtime of this program is ~2 seconds (rather than ~3).

```
timed_thread_context ctx;
std::execution::sender auto a =
  schedule_after(ctx.get_scheduler(), std::chrono::seconds(1)) |
  std::execution::then([]() {
    std::cout << "a" << std::endl;
  });
std::execution::sender auto b =
  schedule_after(ctx.get_scheduler(), std::chrono::seconds(2)) |
  std::execution::then([]() {
    std::cout << "b" << std::endl;
  });
std::this_thread::sync_wait(
  std::execution::when_all(a, b));
```

Let’s invoke a variable number of things

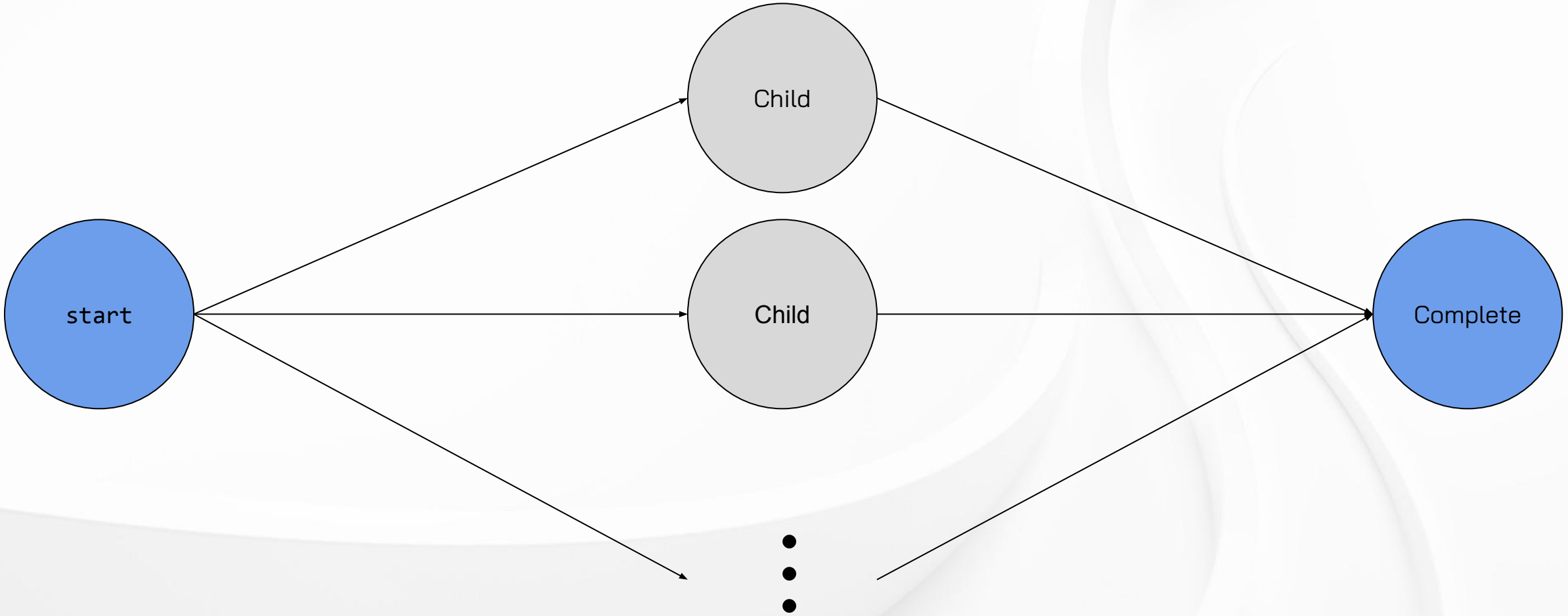


when_all_range

Rather than being a variadic template the input senders are provided via a range, therefore a runtime-determined number of senders may be provided.

```
template<std::ranges::input_range Range>
  requires
    std::execution::sender<
      std::ranges::range_reference_t<Range>> &&
    std::constructible_from<std::decay_t<Range>, Range>
std::execution::sender auto when_all_range(Range&& range);
```

LSEG


Child
start
Child
Complete

when_all_range isn’t in P2300

LSEG


Core C++ 2024
Evolving C++ Networking With Senders & Receivers
Part 1 & 2
Robert Leahy

Synchronous Function Signature

Consider this regular function signature.

```
int foo() noexcept;
```

Synchronous Function Signature

It returns one value, but it could return no values (if it returned `void`).

```
int foo() noexcept;
```

Synchronous Function Signature

It doesn't end in error, but if `noexcept` were missing it could.

```
int foo() noexcept;
```

Asynchronous Function Signature

In the asynchronous domain `foo` would have this signature.

```
std::execution::completion_signatures<
  std::execution::set_value_t(int)>;
```

Synchronous Function Signature

Same function signature again.

```
int foo() noexcept;
```

Synchronous Function Signature

What if it could throw?

```
int foo();
```

Asynchronous Function Signature

Asynchronous functions have different “channels” in the same way that synchronous functions have a return channel and a throw channel.

```
std::execution::completion_signatures<
  std::execution::set_value_t(int),
  std::execution::set_error_t(std::exception_ptr)>;
```

LSEG

Asynchronous Function Signature

In `std::execution` “functions” can return multiple values.

```
std::execution::completion_signatures<
  std::execution::set_value_t(int, float, double),
  std::execution::set_error_t(std::exception_ptr)>;
```

LSEG

Asynchronous Function Signature

In `std::execution` “functions” can return multiple values.

```
std::execution::completion_signatures<
  std::execution::set_value_t(int, float, double),
  std::execution::set_error_t(std::exception_ptr),
  std::execution::set_value_t()>;
```

LSEG

Asynchronous Function Signature

In `std::execution` functions can “return” in a way that regular functions can’t: They can simply stop making forward progress without failing and without completing successfully.

```
std::execution::completion_signatures<
  std::execution::set_value_t(int, float, double),
  std::execution::set_error_t(std::exception_ptr),
  std::execution::set_value_t(),
  std::execution::set_stopped_t()>;
```

Asynchronous Function Signature

How do you get the set of completion signatures from a sender?

```
std::execution::sender auto s = /* ... */;
```

Asynchronous Function Signature

Note that receivers provide an environment which is the second parameter to `completion_signatures_of_t`.

```
std::execution::sender auto s = /* ... */;
using completion_signatures =
  std::execution::completion_signatures_of_t<
    decltype(s),
    std::empty_env>;
```

LSEG

Asynchronous Function

Fully curried function ready to be connected to a receiver.

```
struct sender {

  using sender_concept = std::execution::sender_t;

  template<typename Self, typename Env>
  consteval static
    std::execution::completion_signatures<Signatures...>
    get_completion_signatures();

  template<typename Receiver>
  std::execution::operation_state auto connect(Receiver&&);

};
```

LSEG

Asynchronous Function

What's a "receiver?"

```
struct sender {

  using sender_concept = std::execution::sender_t;

  template<typename Self, typename Env>
  consteval static
    std::execution::completion_signatures<Signatures...>
    get_completion_signatures();

  template<typename Receiver>
  std::execution::operation_state auto connect(Receiver&&);

};
```

LSEG

Asynchronous Return

Asynchronous “functions” “return” by satisfying the “receiver contract” and sending a “completion signal” to a “receiver.”

```
struct receiver {
  using receiver_concept = std::execution::receiver_t;
  void set_value(Ts...) && noexcept;
  void set_error(T) && noexcept;
  void set_stopped() && noexcept;
};
```

Asynchronous Function

What's an "operation state?"

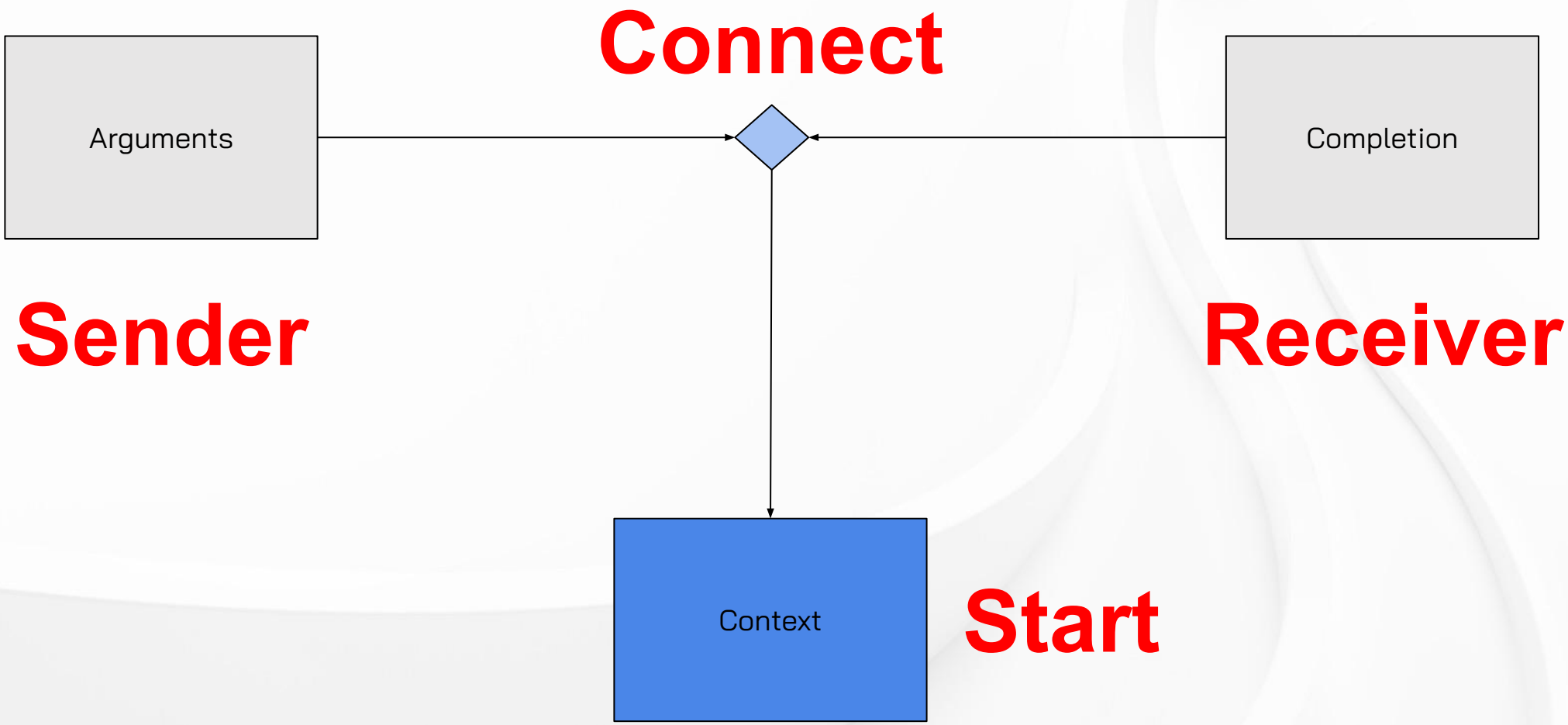
```
struct sender {

  using sender_concept = std::execution::sender_t;

  template<typename Self, typename Env>
  consteval static
    std::execution::completion_signatures<Signatures...>
    get_completion_signatures();

  template<typename Receiver>
  std::execution::operation_state auto connect(Receiver&&);

};
```

LSEG

Asynchronous Stack Frame

Like a synchronous function's stack frame the operation state remains within its lifetime and at a stable location in memory for the duration of the invocation.

```
struct operation_state {

  using operation_state_concept =
    std::execution::operation_state_t;

  void start() & noexcept;

};
```


Connect
Arguments
Completion
Sender
Receiver
Context
Start
Operation State

`when_all`

What is the behavior being complemented?

Does not admit children with multiple value completions

When a child operation completes with

- `set_value` (i.e. success)
  - Store the value(s), if any
- `set_error` (i.e. failure)
  - Store the error, unless another child has already reported an error
  - Issue a stop request to all children
- `set_stopped` (i.e. cancellation)
  - Requests that all children stop

When the last child operation completes

- If all child operations completed with `set_value` sends the concatenation of all values sent thereby, otherwise
- If any child completed with `set_error` sends that error, otherwise
- Sends stopped

when_all_range

What changes from `when_all`?

- All child senders are the same type
  - Since they're elements of a range
- Number of senders can vary at runtime
  - Therefore the number of results can vary at runtime
  - Therefore cannot encapsulate the exact cardinality of results in the type system (as `when_all` does with its `set_value` completion signature)
  - Children might complete with 0, 1, or N values

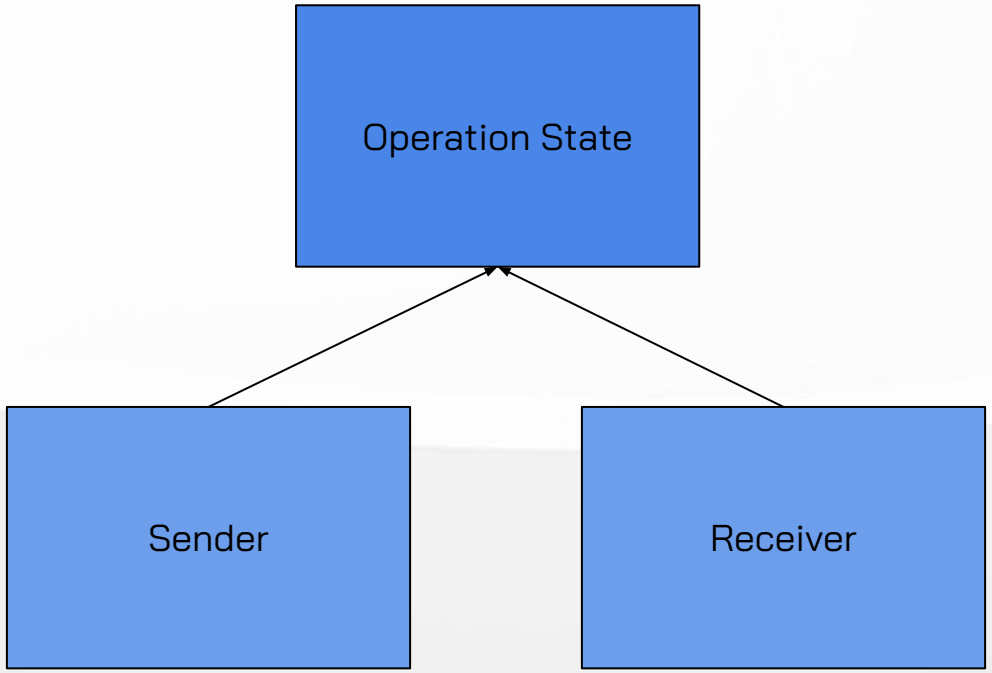
Values	Type
0	`set_value_t()`
1	`set_value_t(std::vector<T>)`
N	`set_value_t(std::vector<std::tuple<Ts...>>)`

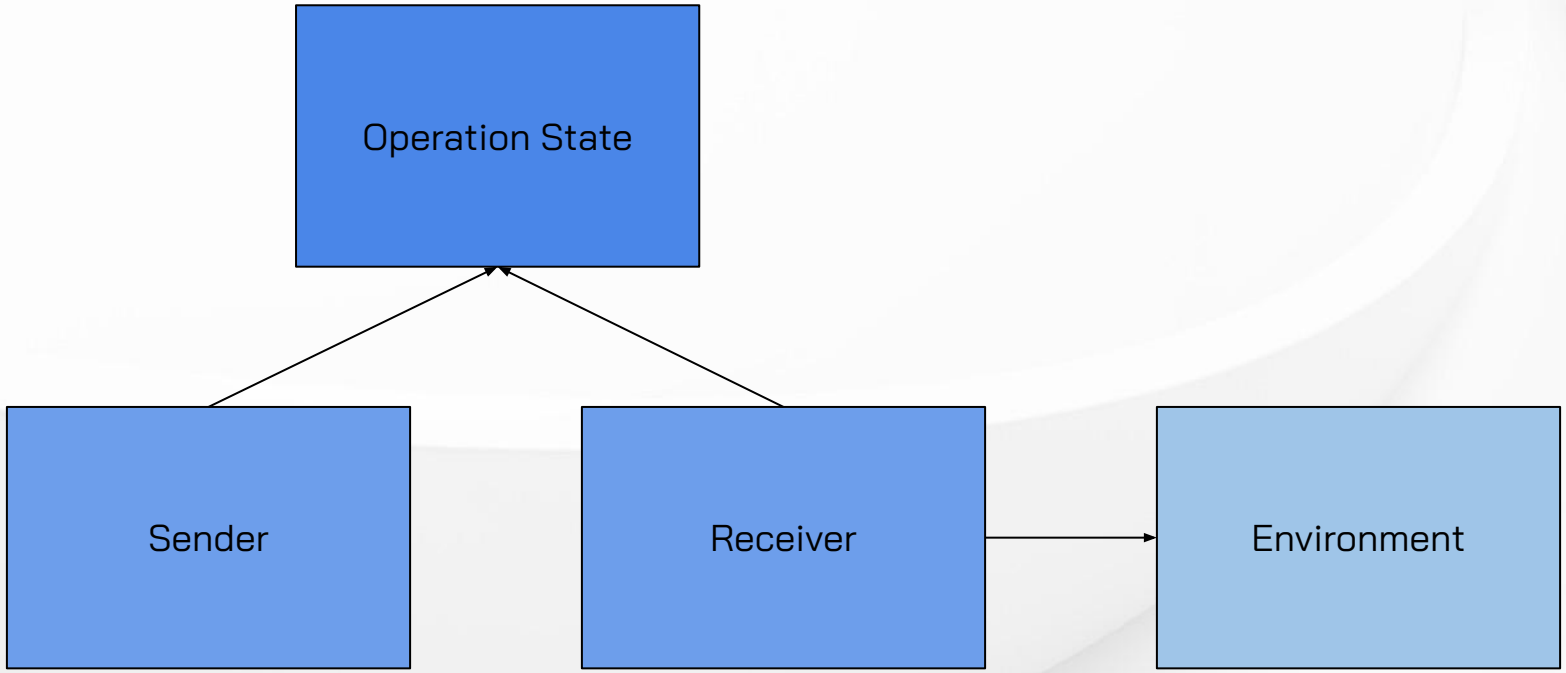
Vector

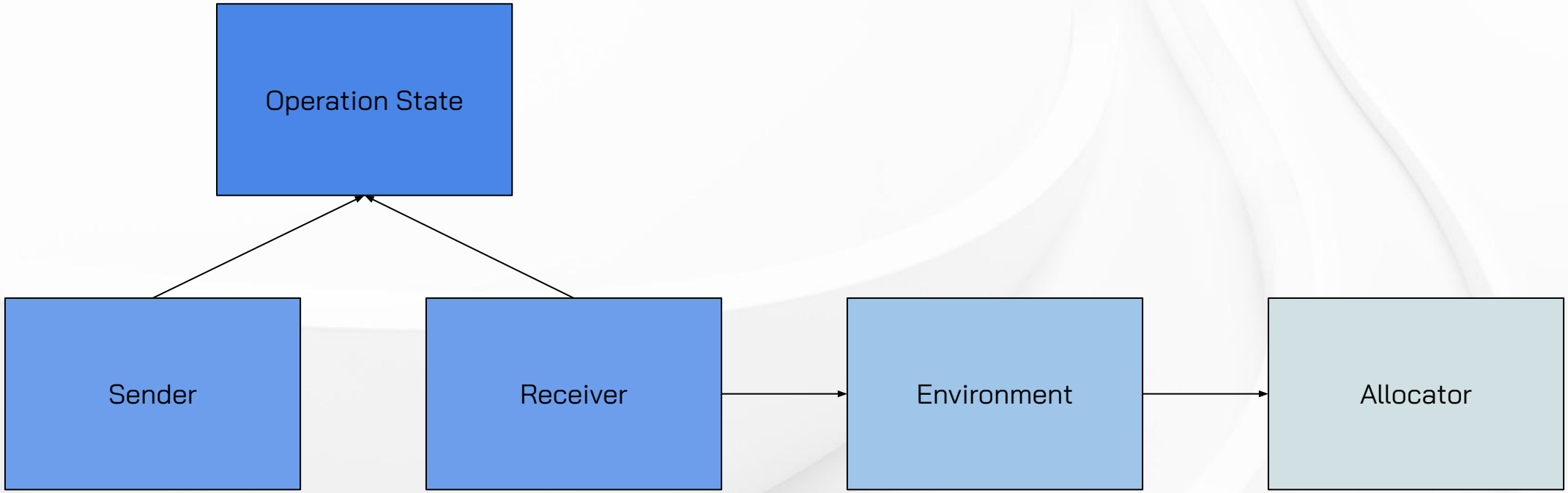
If the children send a single value we want to send a `std::vector<T>`.

If the children send multiple values we want to send a `std::vector<std::tuple<Ts...>>`.

```
template<typename... Args>
struct to_vector {
  using type = std::vector<std::tuple<std::decay_t<Args>...>>;
};

template<typename T>
struct to_vector<T> {
  using type = std::vector<std::decay_t<T>>;
};

template<typename... Args>
using to_vector_t = typename to_vector<Args...>::type;
```


Operation State
Sender
Receiver


Operation State
Sender
Receiver
Environment


Operation State
Sender
Receiver
Environment
Allocator

get_allocator

Provides similar behavior to `std::execution::get_stop_token`, i.e. falls back to a sensible default when the environment doesn't support the query.

```
template<typename Env> requires requires(const Env& e) {
  { std::execution::get_allocator(e) } noexcept;
}
constexpr decltype(auto) get_allocator(const Env& e) noexcept {
  return std::execution::get_allocator(e);
}

template<typename Env>
constexpr std::allocator<int> get_allocator(const Env&) noexcept {
  return {};
}

template<typename Env>
using allocator_of_t = decltype(
  ::get_allocator(std::declval<const Env&>()));
```

get_allocator_for

Standard containers like when you rebind allocators for them.

```
template<typename T, typename Env>
using allocator_for_t =
  typename std::allocator_traits<allocator_of_t<Env>>::
    template rebind_alloc<T>;

template<typename T, typename Env>
constexpr auto get_allocator_for(const Env& e) noexcept {
  return allocator_for_t<T, Env>(::get_allocator(e));
}
```

LSEG

Vector

Now with custom allocator support via an environment.

```
template<typename T, typename Env>
using vector = std::vector<T, allocator_for_t<T, Env>>;

template<typename Env, typename... Args>
struct to_vector {
  using type = vector<std::tuple<std::decay_t<Args>...>, Env>;
};

template<typename Env, typename T>
struct to_vector<Env, T> {
  using type = vector<std::decay_t<T>, Env>;
};

template<typename Env, typename... Args>
using to_vector_t = typename to_vector<Env, Args...>::type;
```

LSEG

Transforming Completion Signatures

Which technique to use?

- Regular template metaprogramming
  - Incredibly verbose
  - Natural solutions involve Boost.Mp11
- `transform_completion_signatures`
  - Out of the box way to work with completion signatures in P2300
  - Terse compared to “regular” template metaprogramming
  - Removed by P3557
- Value-based techniques from P3557
  - Supporting functionality hasn’t been adopted


Reflection

Reflection Utilities

Creates a reflection of a completion signature from a reflection of the tag indicating the completion signal and a range of reflections indicating the types of the arguments.

```
template<typename T, typename... Args>
using make_completion_signature_impl = T(Args...);

template<typename Range = std::initializer_list<std::meta::info>>
consteval auto make_completion_signature(
  const std::meta::info tag,
  Range&& arguments)
{
  std::vector<std::meta::info> v{tag};
  v.append_range(arguments);
  return dealias(substitute(^^make_completion_signature_impl, v));
}
```

LSEG

Completion Signatures

Note the `add` helper which ensures completion signatures are deduplicated.

Stopped completion signatures are copied over unchanged.

Note that `return_type_of` is not from P2996 but rather from P3096.

```
consteval auto completion_signatures(
  const std::meta::info signatures,
  const std::meta::info env)
{
  bool found_value_completion = false;
  std::vector<std::meta::info> results;
  const auto add = [&](std::meta::info info) {
    if (std::ranges::find(results, info) == results.end()) {
      results.push_back(info);
    }
  };
  for (auto&& signature : template_arguments_of(signatures)) {
    const auto tag = return_type_of(signature);
    if (tag == ^^std::execution::set_stopped_t) {
      add(signature);
      continue;
    }
    /* ...
     *
     *
     *
     */
  }
  // ...
}
```

LSEG

Completion Signatures

`parameters_of` is another P3096 function.

We need to be able to store and retrieve all arguments which means decay-copying (to store) and moving (to retrieve).

Therefore we ensure those operations are possible for all arguments, failing to compile if it isn't.

Thereafter we must be able to "throw" (by sending a `std::exception_ptr` on the error channel) if either of those operations can throw.

Lastly we decay each argument to simplify later processing.

LSEG

```
consteval auto completion_signatures(
  const std::meta::info signatures,
  const std::meta::info env)
{
  // ...
  for (auto&& signature : template_arguments_of(signatures)) {
    // ...
    auto arguments = parameters_of(signature);
    for (auto&& argument : arguments) {
      const auto decayed = decay(argument);
      if (!(
        is_constructible_type(decayed, {argument}) &&
        is_move_constructible_type(decayed)))
          throw /* ... */;
      if (!(
        is_nothrow_constructible_type(decayed, {argument}) &&
        is_nothrow_move_constructible_type(decayed)))
          add(^^std::execution::set_error_t(std::exception_ptr));
      argument = decayed;
    }
    /* ...
     */
  }
  // ...
}
```

Completion Signatures

If the current signature is an error signature we add it and move on.

```
consteval auto completion_signatures(
  const std::meta::info signatures,
  const std::meta::info env)
{
  // ...
  for (auto&& signature : template_arguments_of(signatures)) {
    // ...
    if (tag == ^^std::execution::set_error_t) {
      add(make_completion_signature(tag, arguments));
      continue;
    }
    /* ...
     *
     *
     *
     *
     *
     *
     *
     *
     *
     *
     */
  }
  // ...
}
```

LSEG

Completion Signatures

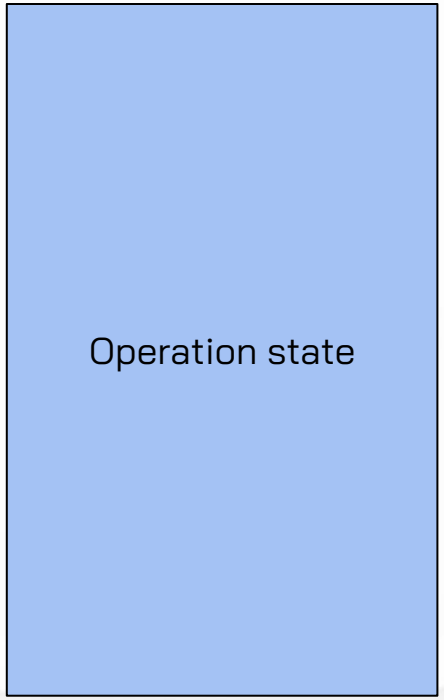
Multiple value completions are disallowed by throwing an exception.

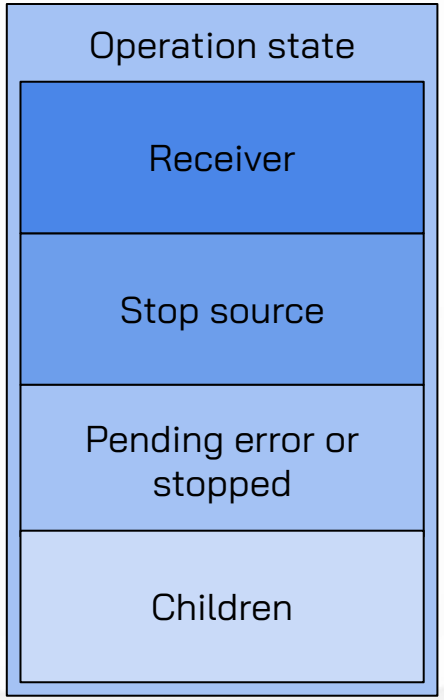
No arguments means the signature is copied over.

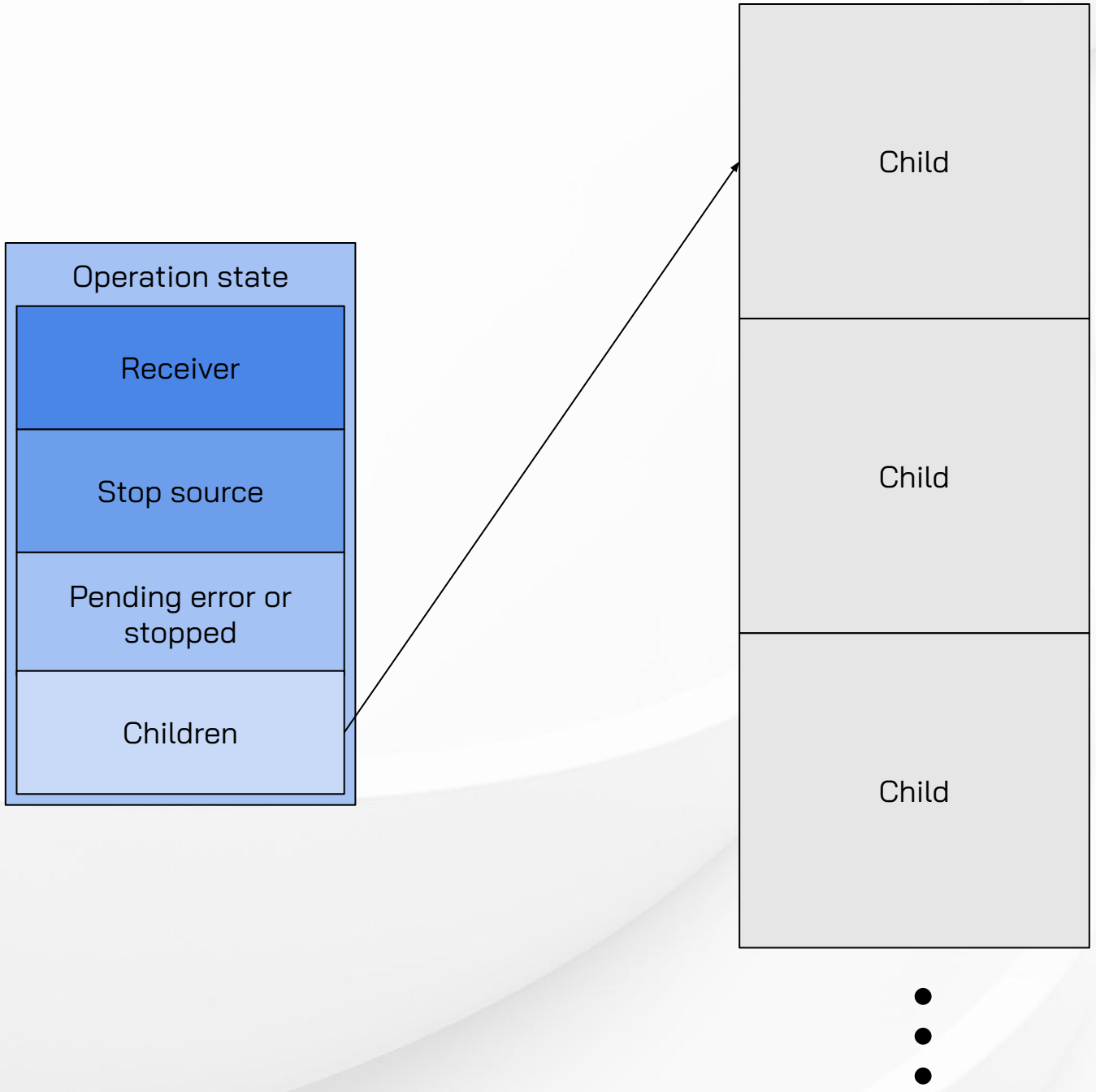
Otherwise we obtain a value completion signature which only sends an appropriate `std::vector`.

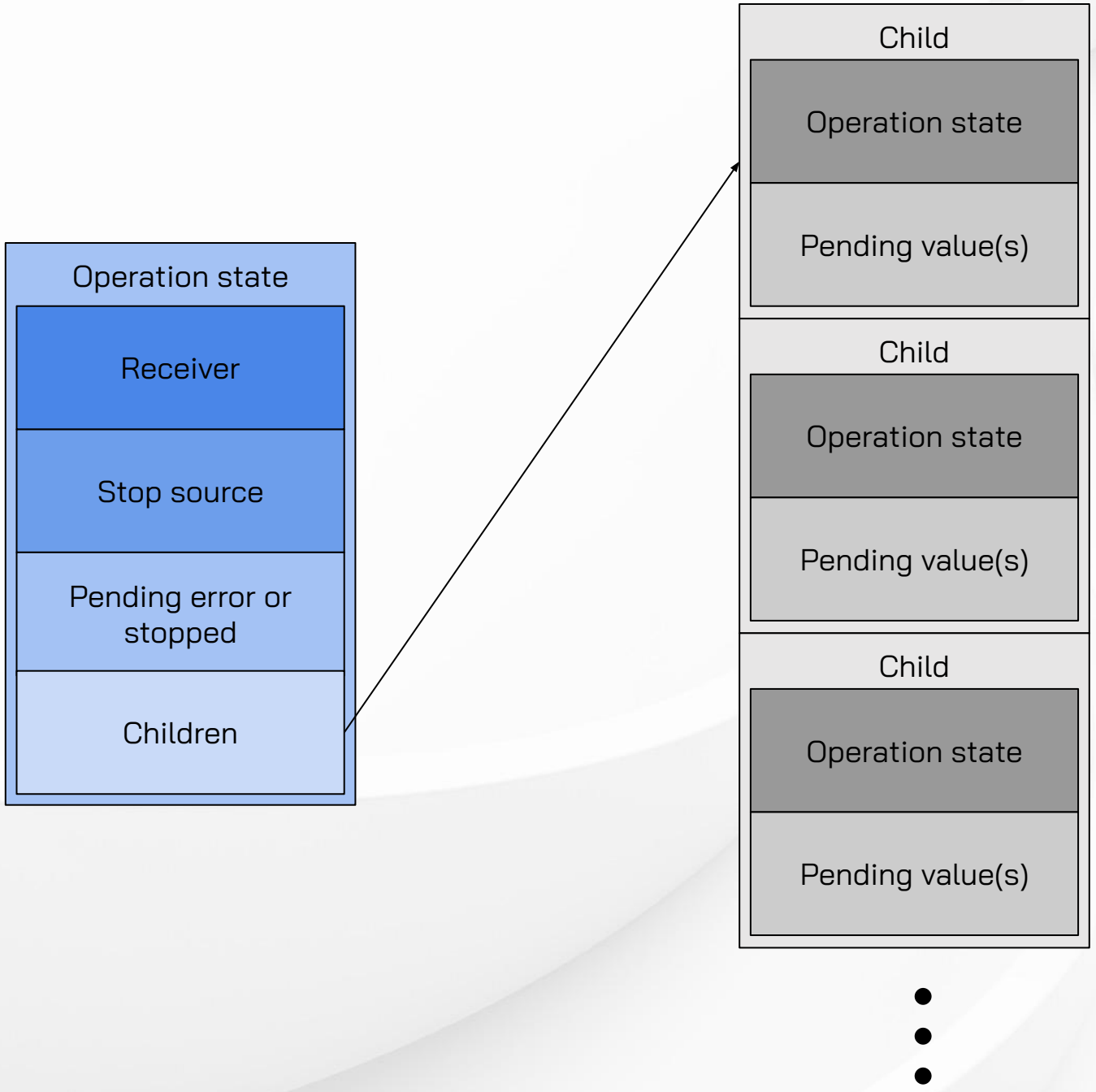
```
consteval auto completion_signatures(
  const std::meta::info signatures,
  const std::meta::info env)
{
  // ...
  for (auto&& signature : template_arguments_of(signatures)) {
    // ...
    if (found_value_completion) {
      throw /* ... */;
    }
    found_value_completion = true;
    if (arguments.empty()) {
      add(^^std::execution::set_value_t());
      continue;
    }
    arguments.insert(arguments.begin(), env);
    add(
      make_completion_signature(
        tag,
        {dealias(substitute(^^to_vector_t, arguments))}));
  }
  return substitute(
    ^^std::execution::completion_signatures,
    results);
}
```

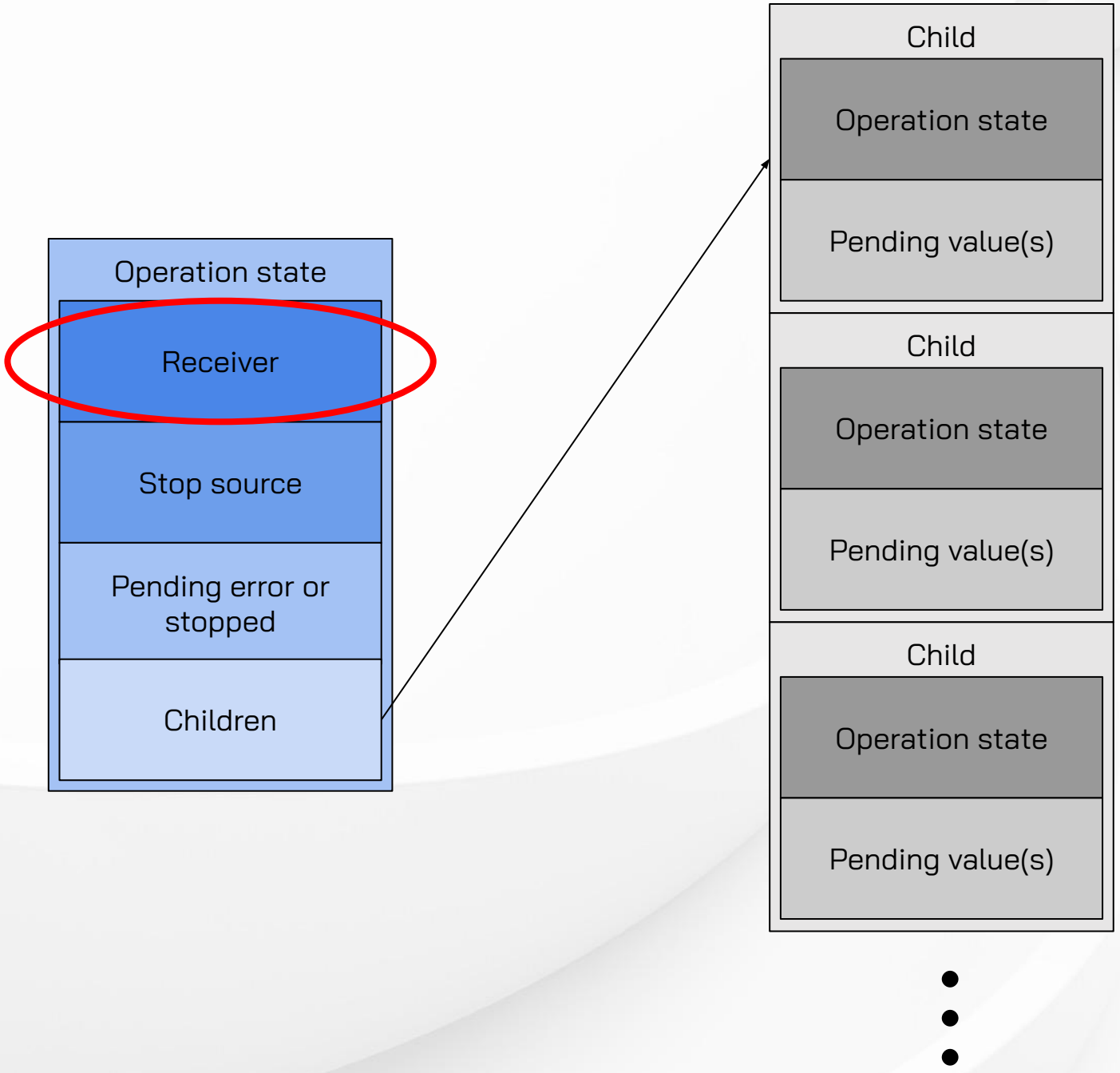
LSEG

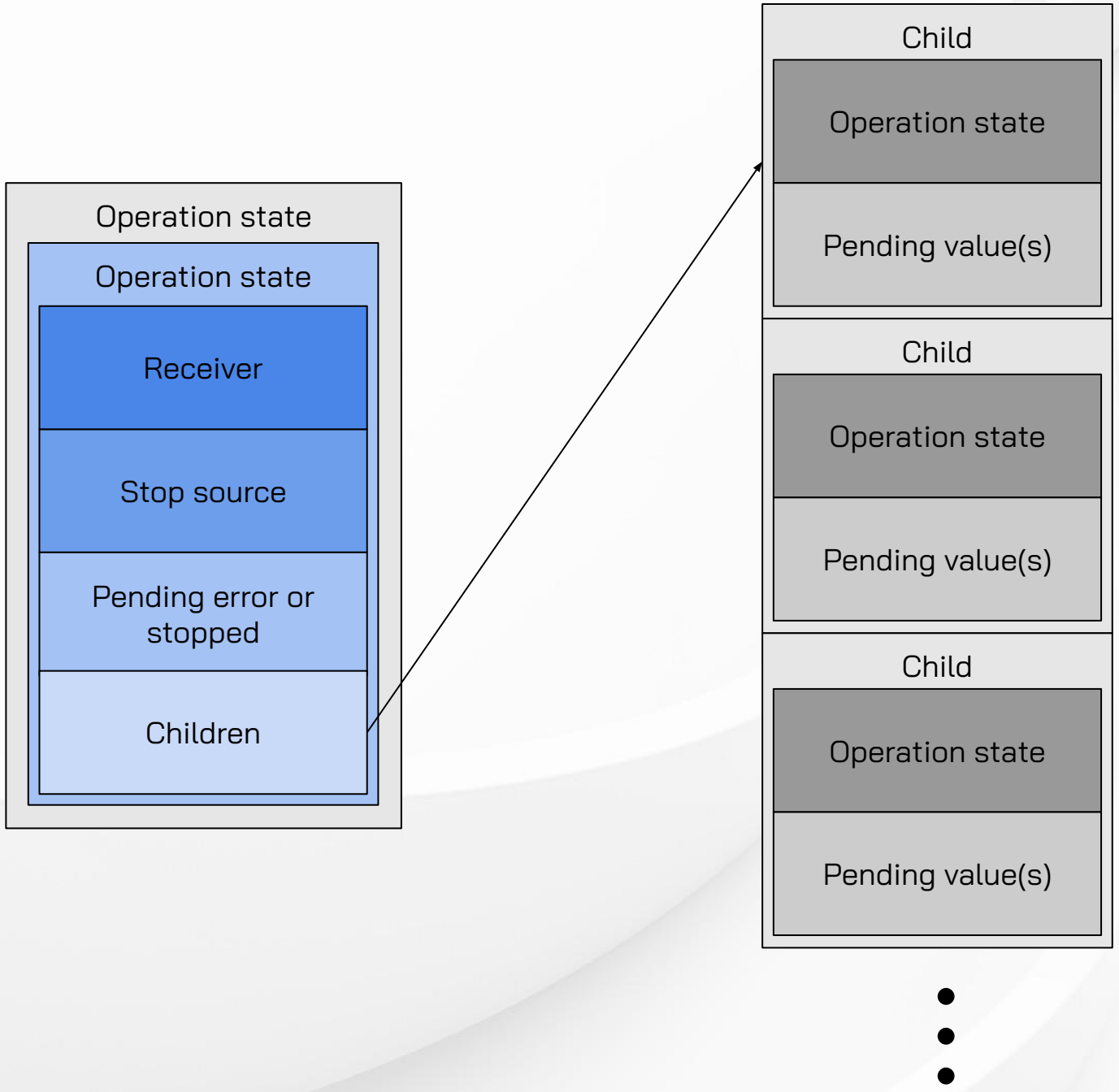

Operation state

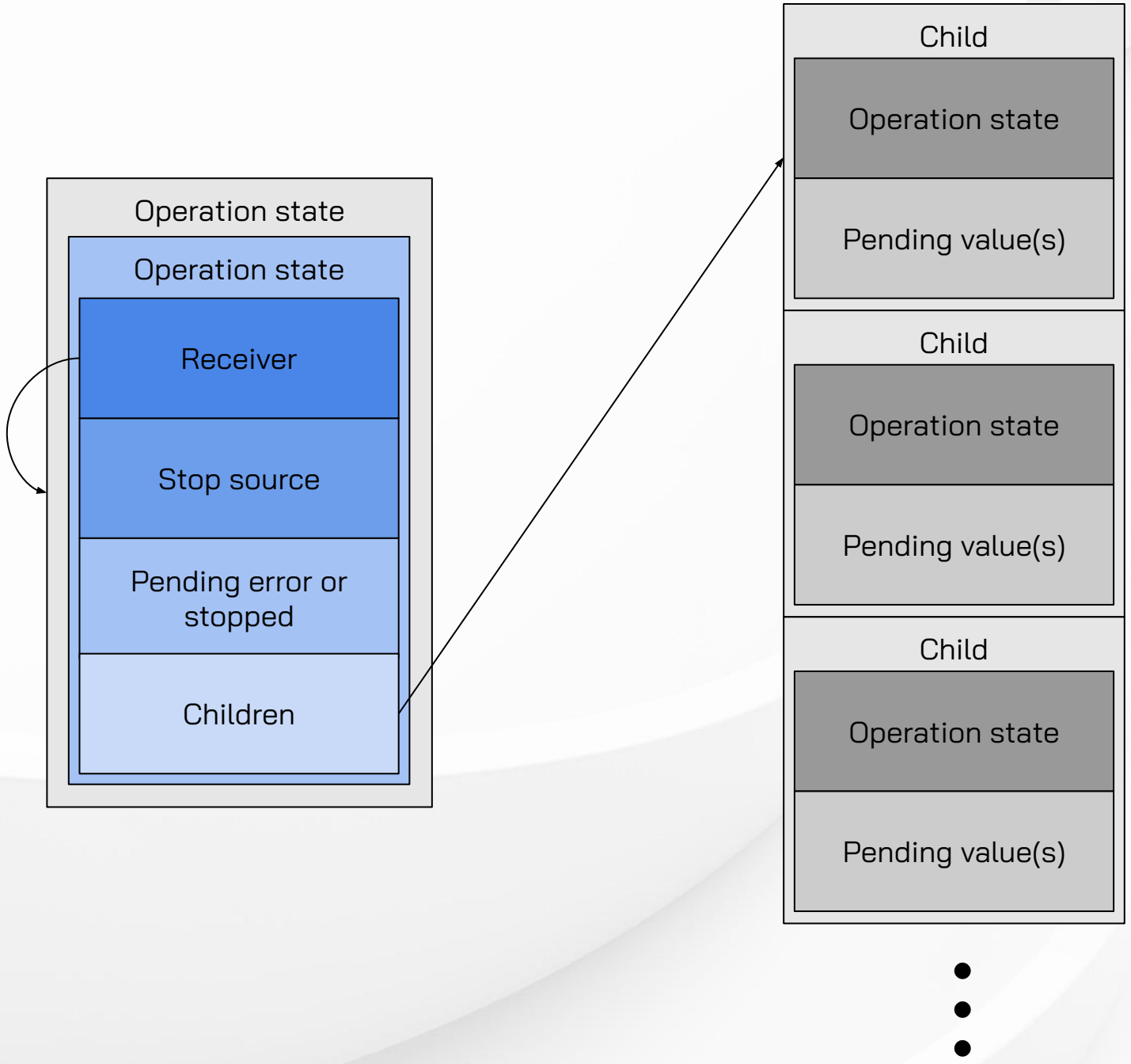

Operation state
Receiver
Stop source
Pending error or stopped
Children


Operation state
Receiver
Stop source
Pending error or stopped
Children
Child
Child
Child


Operation state
Receiver
Stop source
Pending error or stopped
Children
Child
Operation state
Pending value(s)
Child
Operation state
Pending value(s)
Child
Operation state
Pending value(s)


Operation state
Receiver
Stop source
Pending error or stopped
Children
Child
Operation state
Pending value(s)
Child
Operation state
Pending value(s)
Child
Operation state
Pending value(s)


Operation state
Operation state
Receiver
Stop source
Pending error or stopped
Children
Child
Operation state
Pending value(s)
Child
Operation state
Pending value(s)
Child
Operation state
Pending value(s)


Operation state
Operation state
Receiver
Stop source
Pending error or stopped
Children
Child
Operation state
Pending value(s)
Child
Operation state
Pending value(s)
Child
Operation state
Pending value(s)

D3425R1: Reducing operation-state sizes for subobject child operations

Table of Contents

inlinable_receiver

Core of the protocol added by P3425. Allows receivers to be synthesized on the fly from the address of the operation state which would otherwise contain them.

Note that `ChildOp` is not constrained by `operation_state` due to the fact that would require the operation state type to be complete at the point where `inlinable_receiver` is evaluated.

```
template<typename Rcvr, typename ChildOp>
concept inlinable_receiver =
  std::execution::receiver<Rcvr> &&
  requires (ChildOp* op) {
    { Rcvr::make_receiver_for(op) } noexcept
      -> std::same_as<Receiver>;
  };
```

Storing the Receiver

CRTP base class which actually stores the receiver if it isn't "inlinable."

A specialization will deal with the case where the receiver is inlinable (next slide).

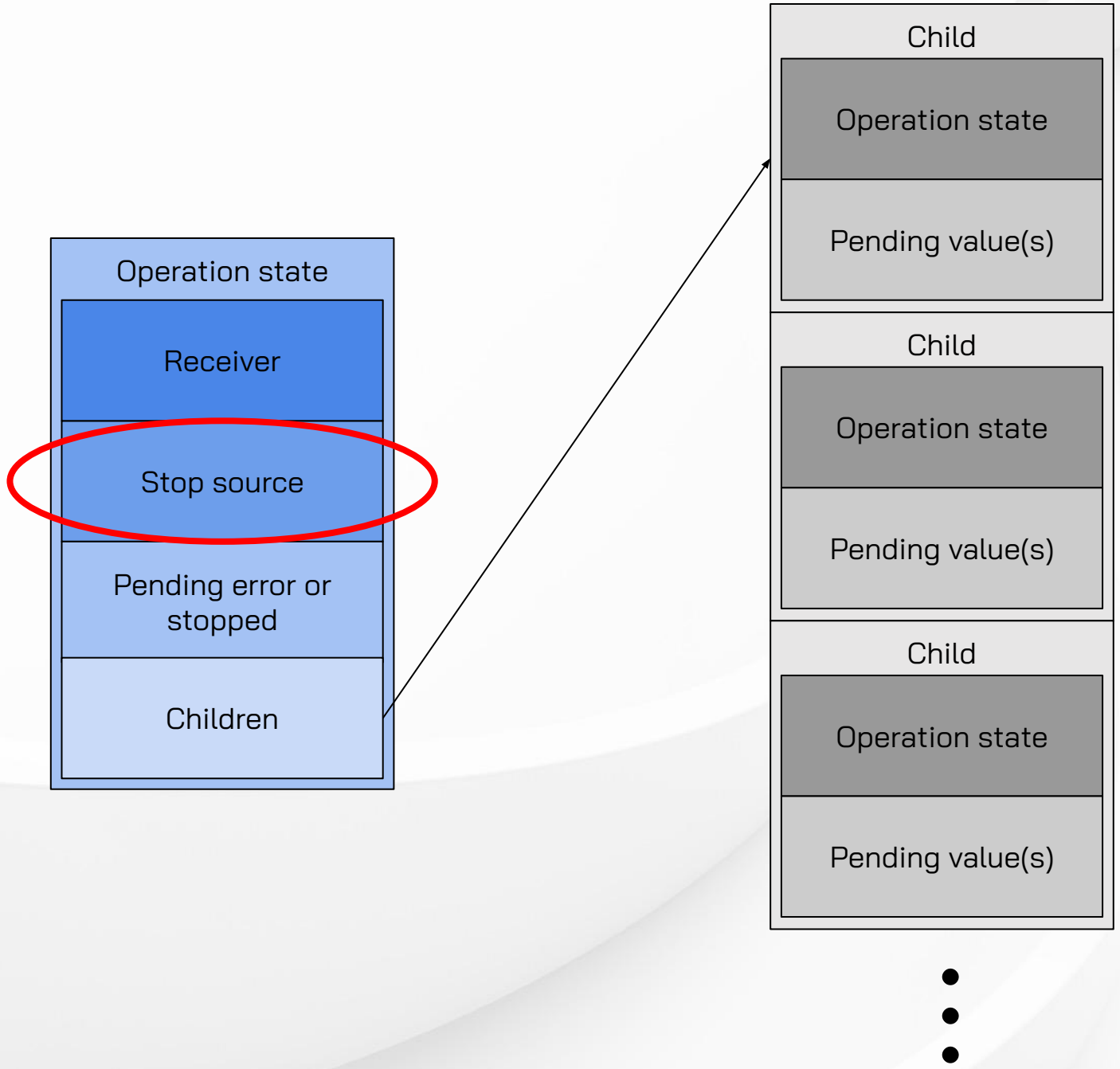
This template is not proposed by P3425 but is described therein (and may be proposed for standardization later, separately).

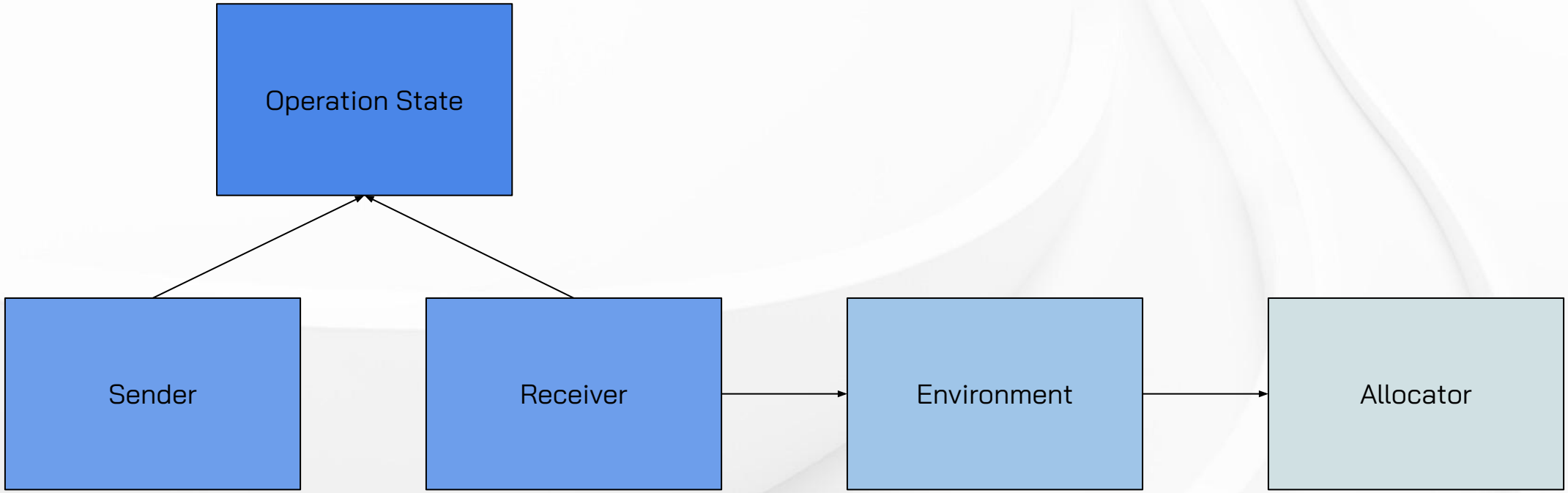
```
template<typename Derived, std::execution::receiver Receiver>
class inlinable_operation_state {
  Receiver rcvr_;
protected:
  constexpr explicit inlinable_operation_state(Receiver r)
    noexcept(std::is_nothrow_move_constructible_v<Receiver>)
    : rcvr_(std::move(r))
  {}
  constexpr Receiver& get_receiver() noexcept {
    return rcvr_;
  }
};
```

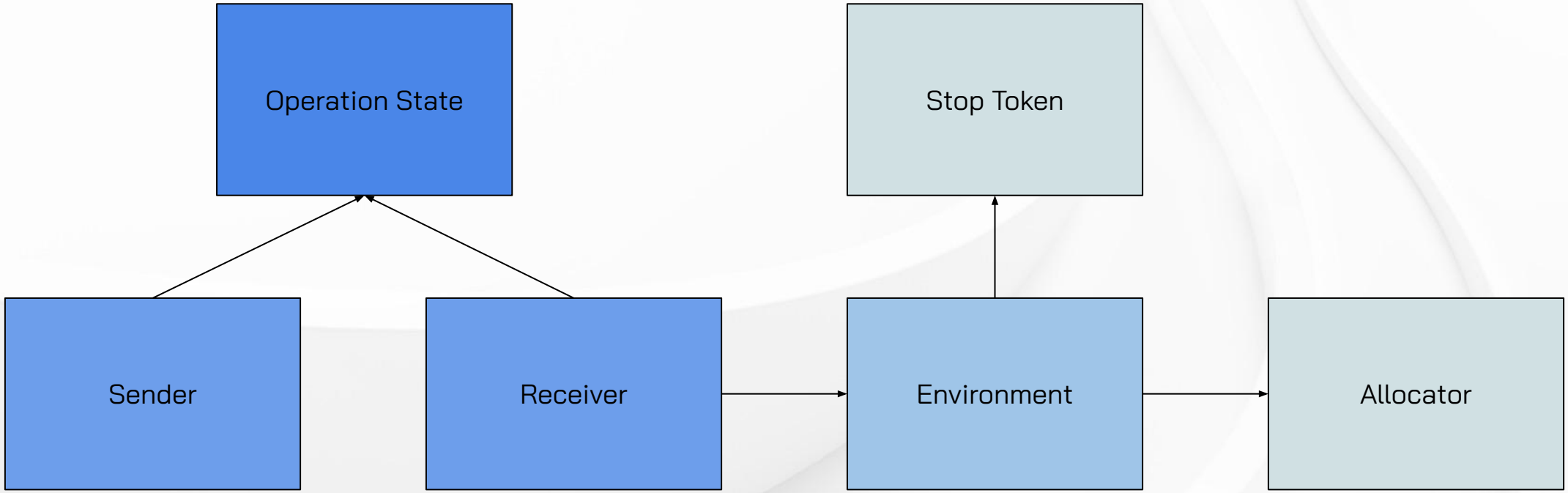
Storing the Receiver

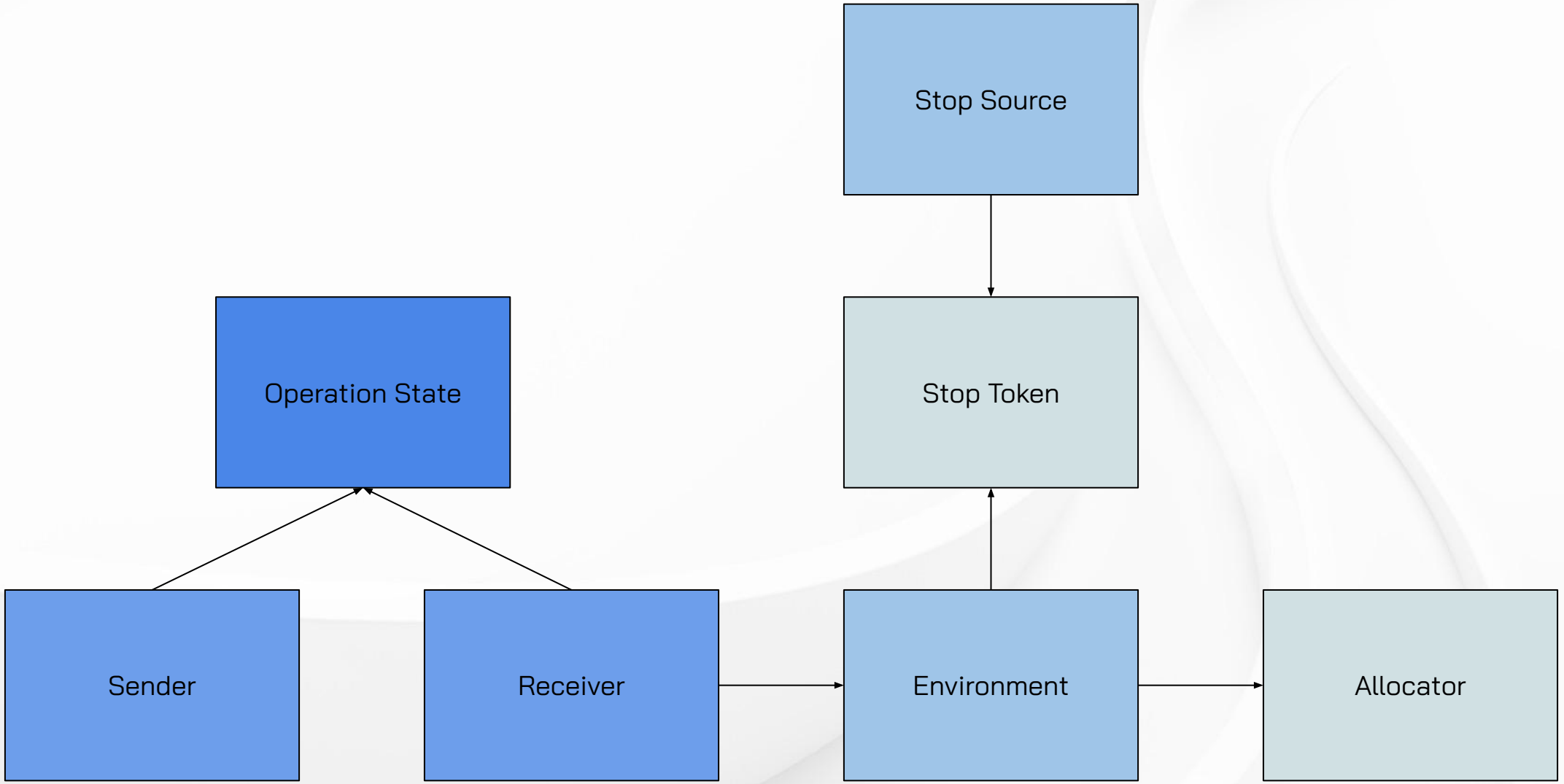
If the receiver is inlinable there's no reason to store it and a prvalue receiver is synthesized whenever needed.

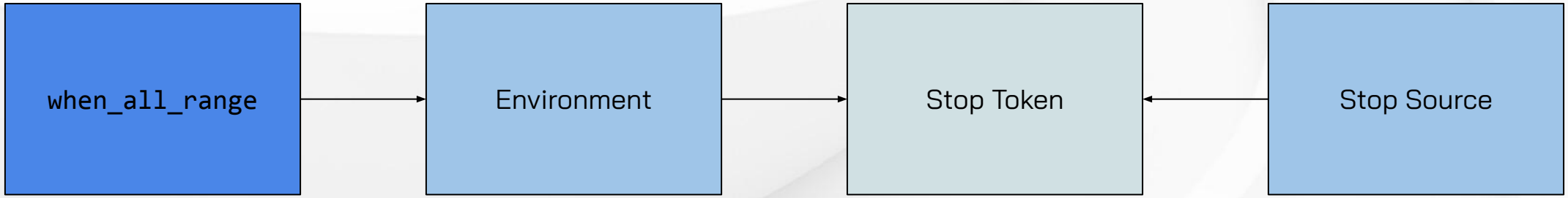
```
template<typename Derived, std::execution::receiver Receiver>
  requires std::execution::inlinable_receiver<
    Receiver,
    Derived>
class inlinable_operation_state<Derived, Receiver> {
protected:
  constexpr explicit inlinable_operation_state(const Receiver&)
    noexcept
  {}
  constexpr Receiver get_receiver() noexcept {
    return Receiver::make_receiver_for(
      static_cast<Derived*>(this));
  }
};
```

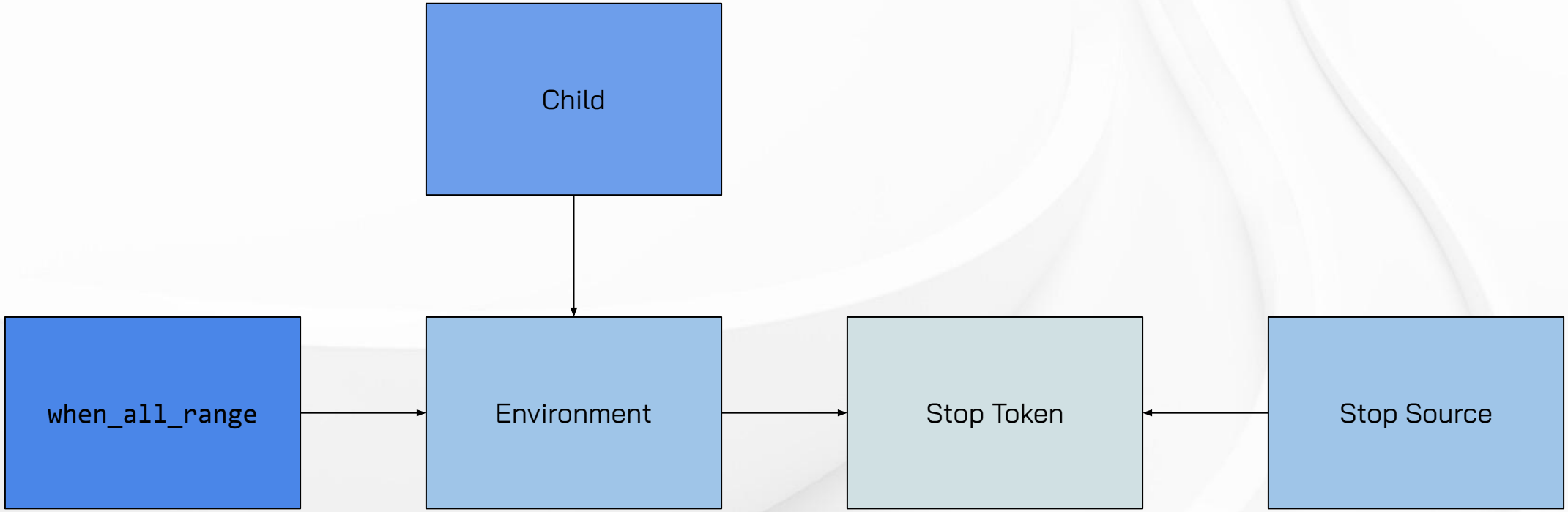

Operation state
Receiver
Stop source
Pending error or stopped
Children
Child
Operation state
Pending value(s)
Child
Operation state
Pending value(s)
Child
Operation state
Pending value(s)

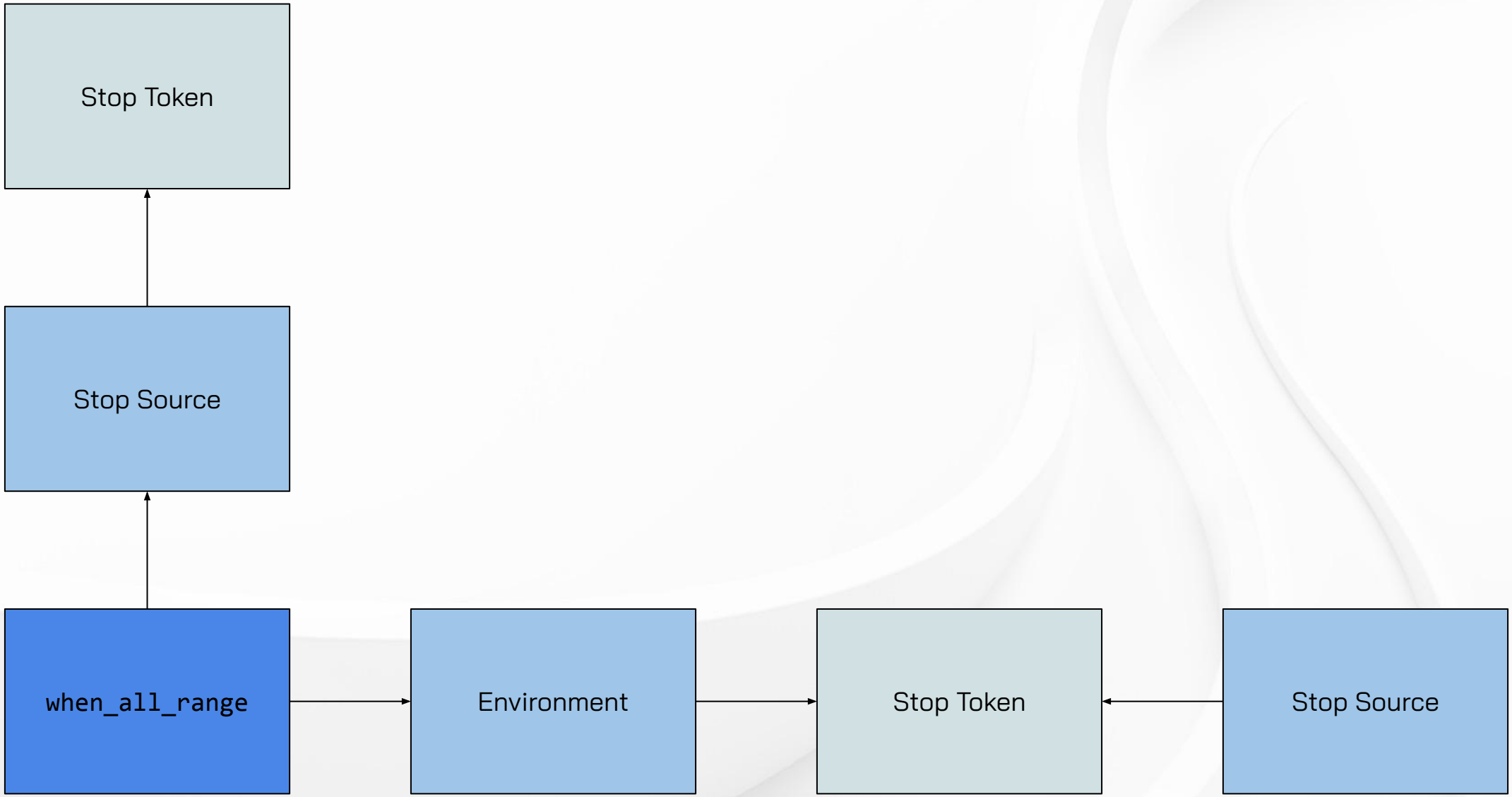

Operation State
Sender
Receiver
Environment
Allocator

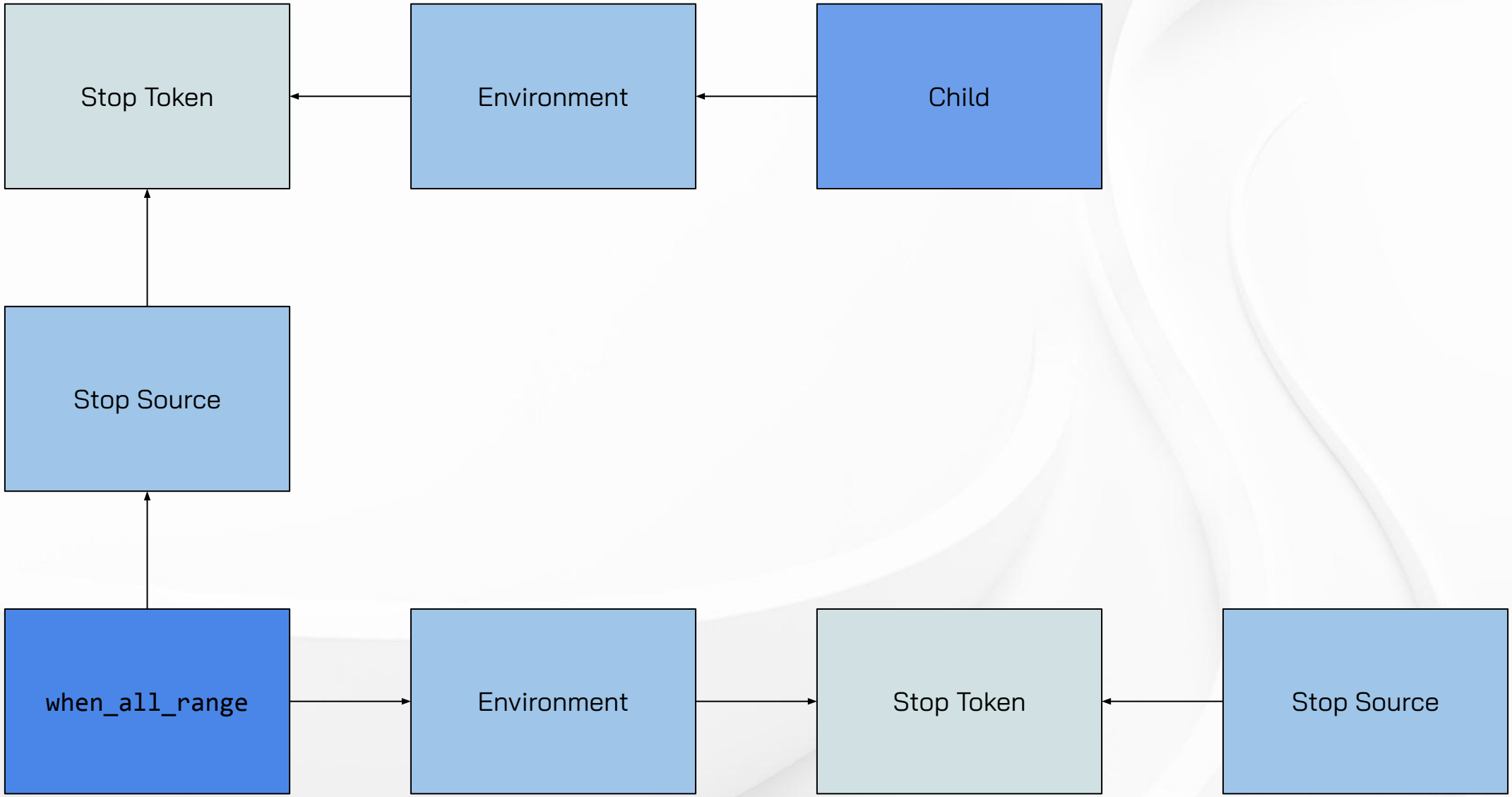

Operation State
Stop Token
Sender
Receiver
Environment
Allocator

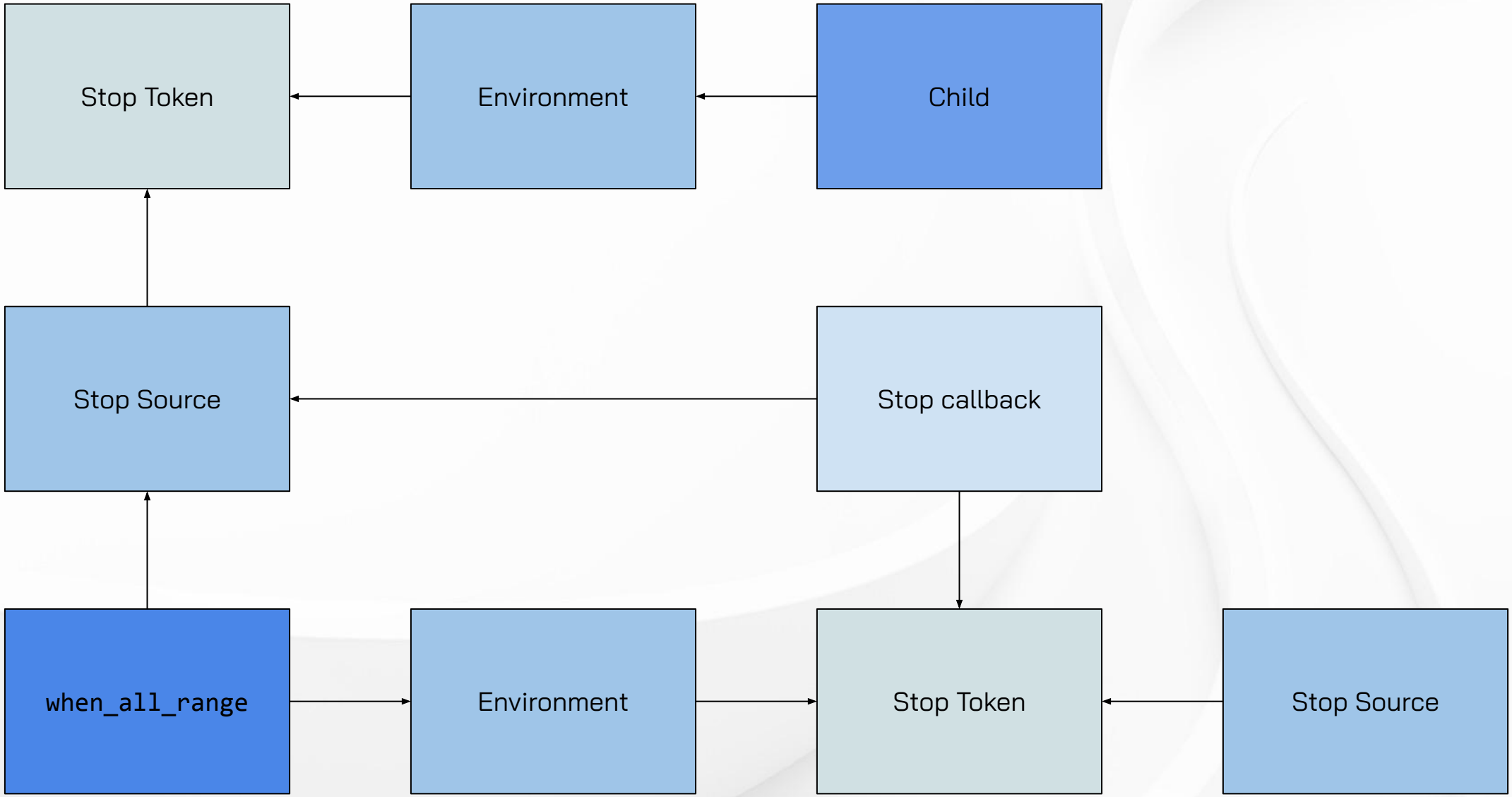

Stop Source
Operation State
Stop Token
Sender
Receiver
Environment
Allocator

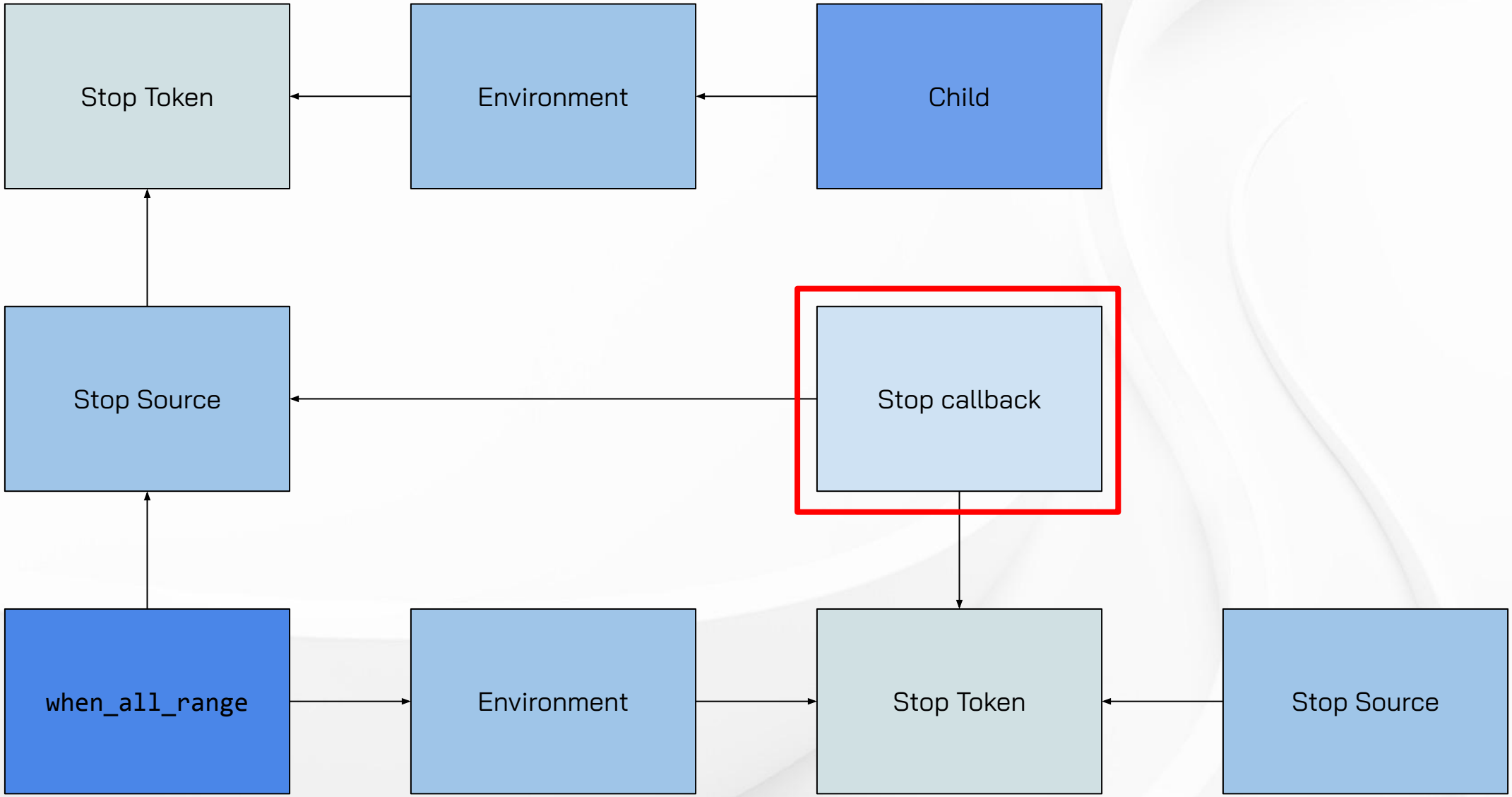

when_all_range
Environment
Stop Token
Stop Source


Child
when_all_range
Environment
Stop Token
Stop Source


Stop Token
Stop Source
when_all_range
Environment
Stop Token
Stop Source


Stop Token
Environment
Child
Stop Source
when_all_range
Environment
Stop Token
Stop Source

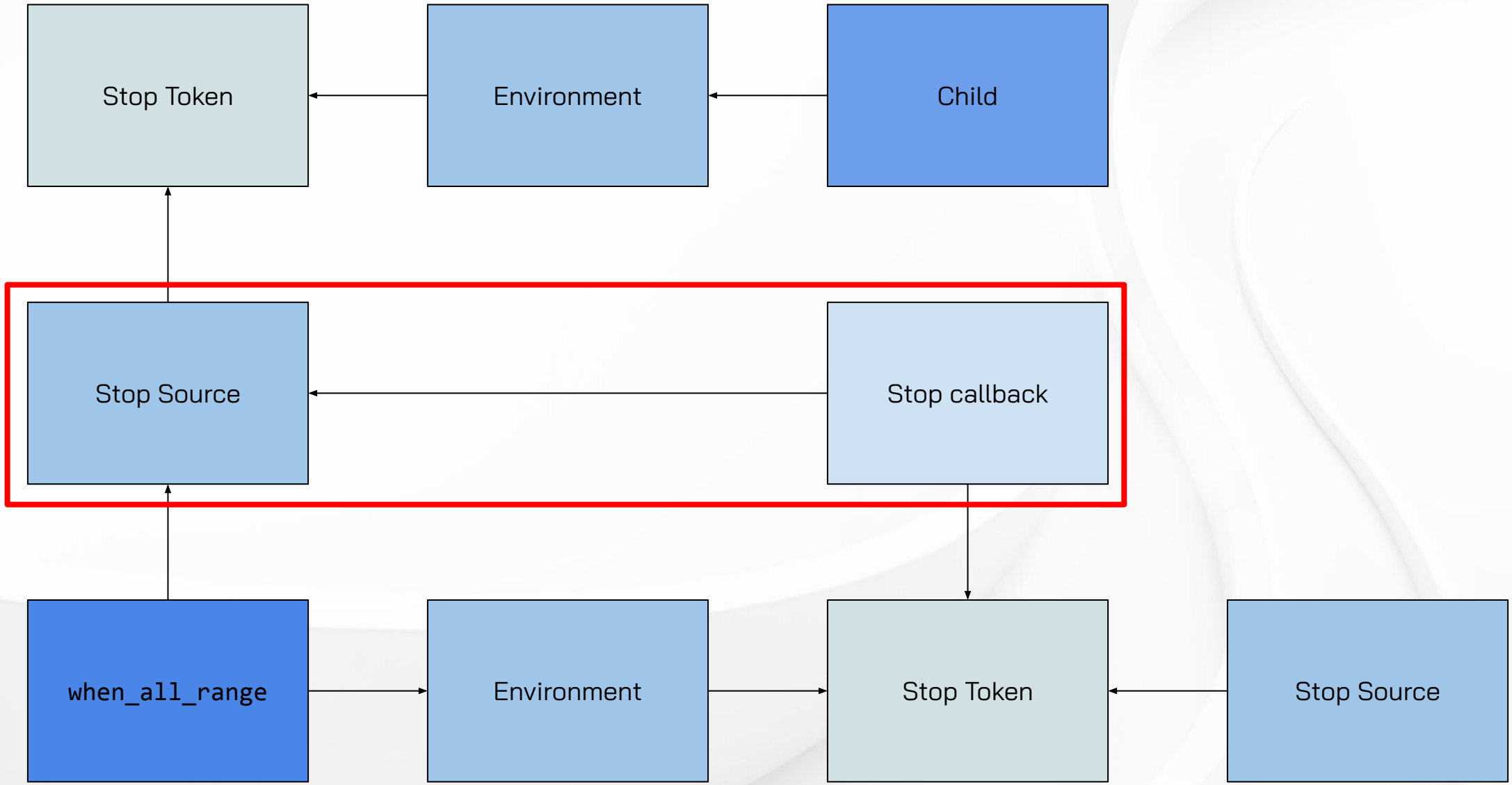

Stop Token
Environment
Child
Stop Source
Stop callback
when_all_range
Environment
Stop Token
Stop Source


Stop Token
Environment
Child
Stop Source
Stop callback
when_all_range
Environment
Stop Token
Stop Source

on_stop_request

When invoked re-emits the invocation as a stop request against its associated `std::inplace_stop_source`.

```
struct on_stop_request {
  constexpr explicit on_stop_request(
    std::inplace_stop_source& source) noexcept
    : source_(source)
  {}
  void operator()() && noexcept {
    source_.request_stop();
  }
private:
  std::inplace_stop_source& source_;
};
```


Stop Token
Environment
Child
Stop Source
Stop callback
when_all_range
Environment
Stop Token
Stop Source

chained_stop_source

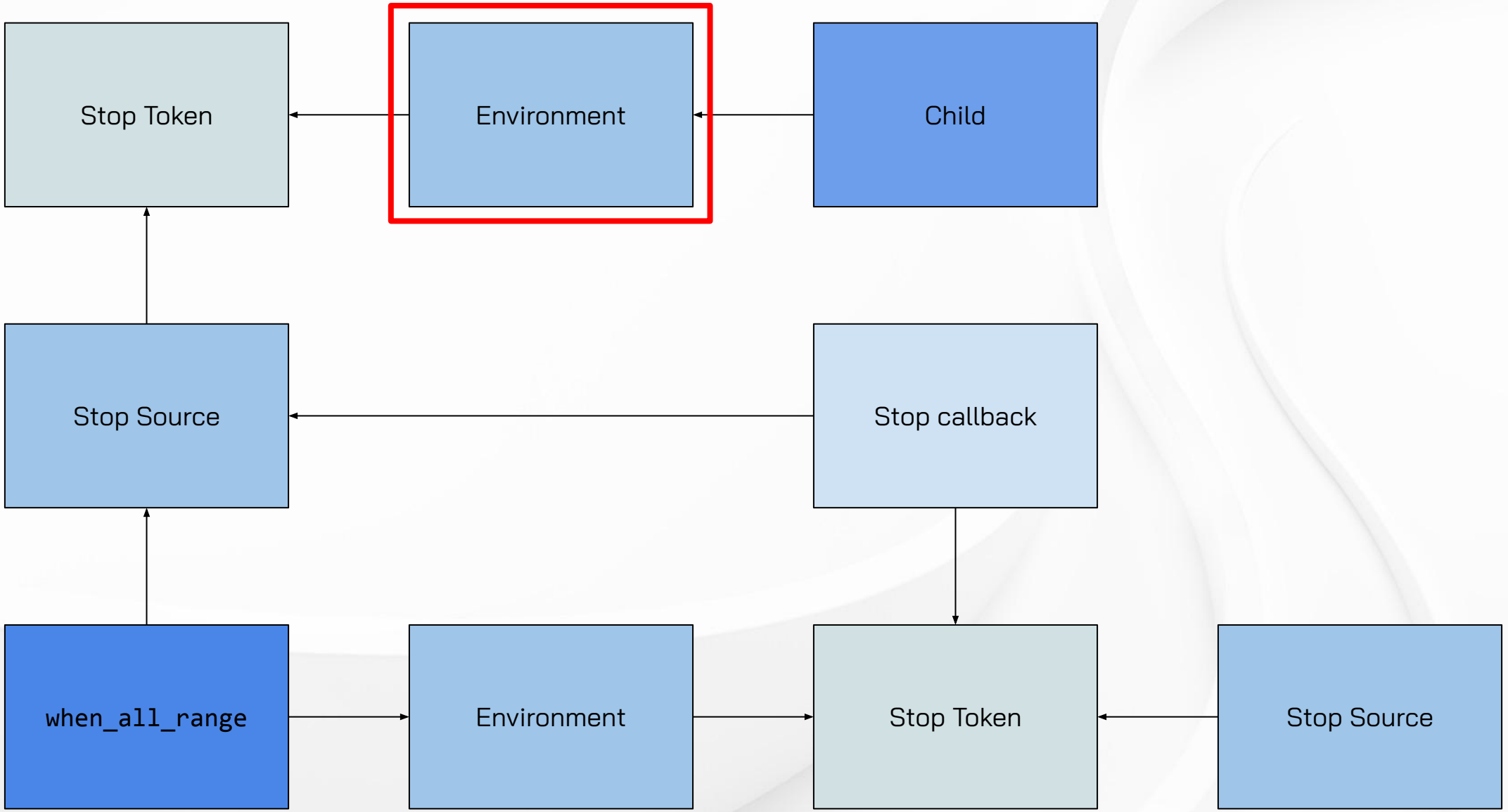
Provides a stop source (by deriving from `std::inplace_stop_source`) allowing us to emit our own stop requests.

Also installs a stop callback against a downstream stop source which propagates stop requests submitted therethrough to our stop source.

This allows us to emit stop requests while also receiving stop requests provided through the stop source associated with our receiver.

```
template<std::stoppable_token Token>
class chained_stop_source : public std::inplace_stop_source {
  using callback_type_ = std::stop_callback_for_t<
    Token,
    on_stop_request>;
  callback_type_ callback_;
public:
  constexpr explicit chained_stop_source(
    const Token& token) noexcept(/* ... */)
    : callback_(
        token,
        static_cast<std::inplace_stop_source&>(*this))
  {}
};
```

LSEG

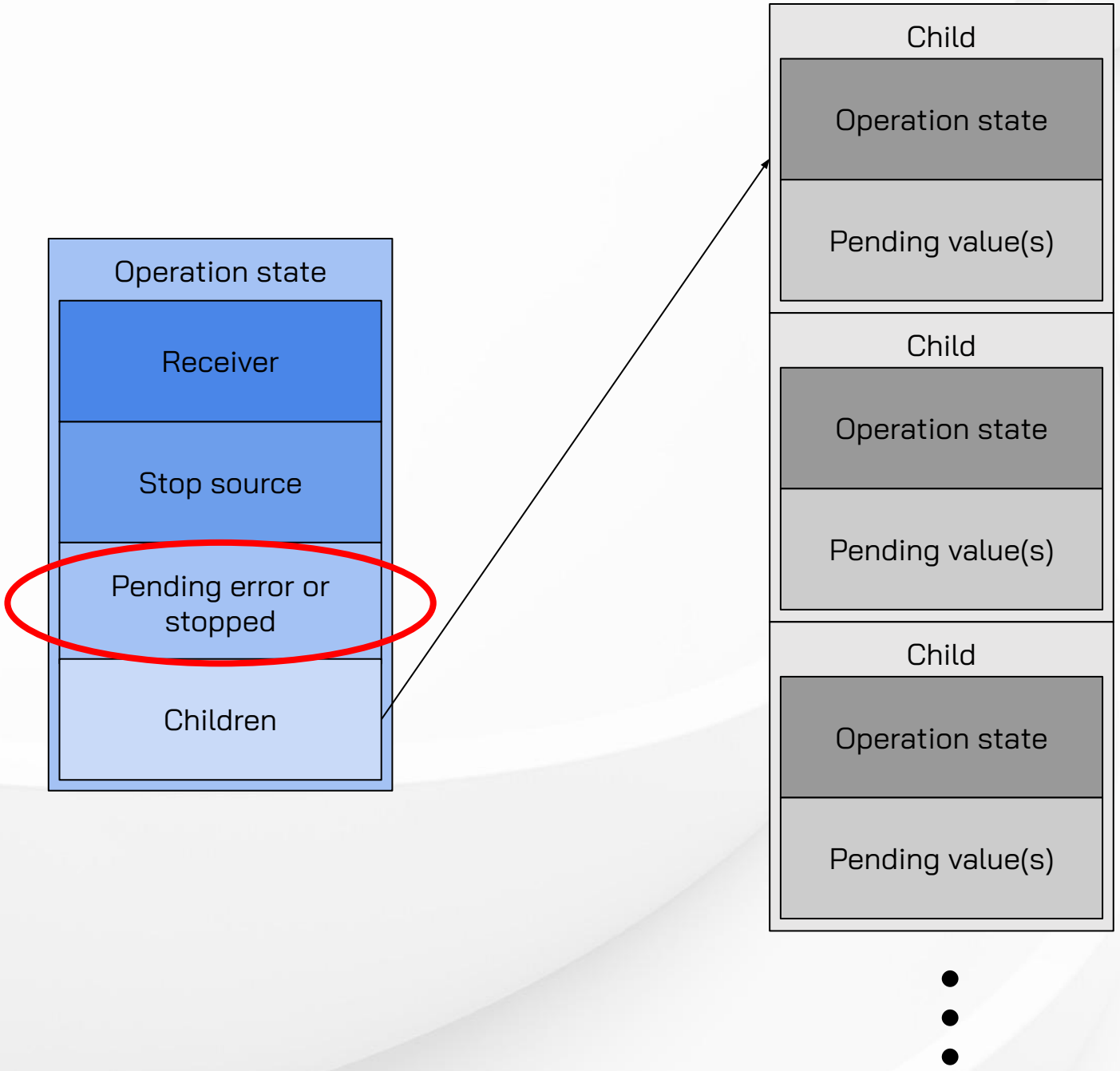

Stop Token
Environment
Child
Stop Source
Stop callback
when_all_range
Environment
Stop Token
Stop Source

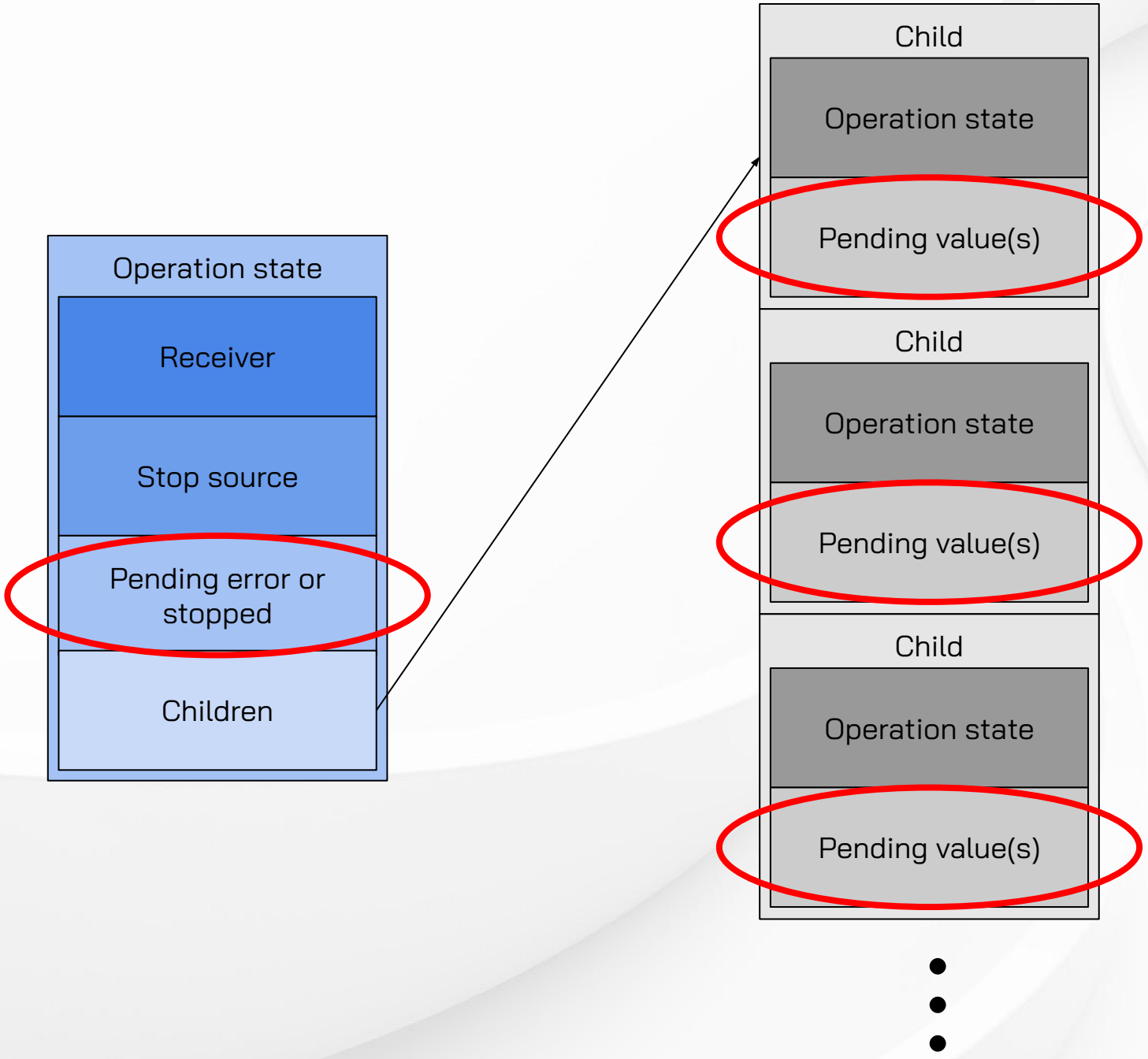
stop_source_env

Wraps an environment and passes through all its queries except `get_stop_token` which is redirected to return a stop token from an alternate stop source.

```
template<typename Env>
struct stop_source_env {
  constexpr explicit stop_source_env(
    std::inplace_stop_source& source,
    Env env) noexcept(/* ... */)
    : source_(source),
      env_(std::move(env))
  {}
  constexpr decltype(auto) query(
    const std::execution::get_stop_token_t&) const noexcept
  {
    return source_.get_token();
  }
  template<typename T>
    requires requires(const Env& env, const T& t) {
      env.query(t);
    }
  constexpr decltype(auto) query(const T& t) const noexcept {
    return env_.query(t);
  }
private:
  std::inplace_stop_source& source_;
  Env env_;
};
```

LSEG


Operation state
Receiver
Stop source
Pending error or stopped
Children
Child
Operation state
Pending value(s)
Child
Operation state
Pending value(s)
Child
Operation state
Pending value(s)


Operation state
Receiver
Stop source
Pending error or stopped
Children
Child
Operation state
Pending value(s)
Child
Operation state
Pending value(s)
Child
Operation state
Pending value(s)

Completion Storage

`arrive` stores the completion (tag indicating the channel, values associated therewith).

`visit` calls the supplied visitor with the stored completion, if any. If there's a stored completion (and the visitor was invoked) `true` is returned, otherwise `false`.

```
struct storage {

  template<typename Tag, typename... Args>
  void arrive(const Tag&, Args&&...);

  template<typename Visitor>
  bool visit(Visitor&&) &&;

};
```

Completion Storage

Version for empty set of completion signatures doesn't even have `arrive` because it catches the scenario where there are no relevant completion signatures (and therefore there should never be a call to `arrive`).

`visit` unconditionally returns `false` because no completions can be stored.

```
template<typename>
struct storage_for_completion_signatures;

template<>
struct storage_for_completion_signatures<
  std::execution::completion_signatures<>>
{
  template<typename Visitor>
  constexpr bool visit(const Visitor&) && noexcept {
    return false;
  }
};
```

Completion Storage

Transforms a completion signature to a tuple which can be stored.

```
template<typename>
struct signature_to_tuple;

template<typename Tag, typename... Args>
struct signature_to_tuple<Tag(Args...)> {
  using type = std::tuple<Tag, std::decay_t<Args>...>;
};
```

LSEG

Completion Storage

Our storage is a variant which starts holding no completion, and which can hold any of the completions for which this type represents storage.

```
template<typename... Signatures>
class storage_for_completion_signatures<
  std::execution::completion_signatures<Signatures...>>
{
  using storage_type_ = ::boost::mp11::mp_unique<
    std::variant<
      std::monostate,
      typename signature_to_tuple<Signatures>::type...>>;
  storage_type_ storage_;
  /* ...
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   */
};
```

LSEG

Completion Storage

If emplace throws the variant will be left “valueless by exception” therefore it’s important we catch the exception and reset the stored alternative back to `std::monostate`.

```
template<typename... Signatures>
class storage_for_completion_signatures</* ... */> {
  // ...
public:
  template<typename... Args>
  constexpr void arrive(Args&&... args) noexcept(
    std::is_nothrow_constructible_v<
      std::tuple<std::decay_t<Args>...>,
      Args...>)
  {
    try {
      storage_.template emplace<
        std::tuple<std::decay_t<Args>...>>(
          std::forward<Args>(args)...);
    } catch (...) {
      storage_ = std::monostate{};
      throw;
    }
  }
  // ...
};
```

Visit Without Throwing?

Whether or not something throws really matters because in `std::execution` the only way out is down: Exceptions need to be caught and sent down the error channel.

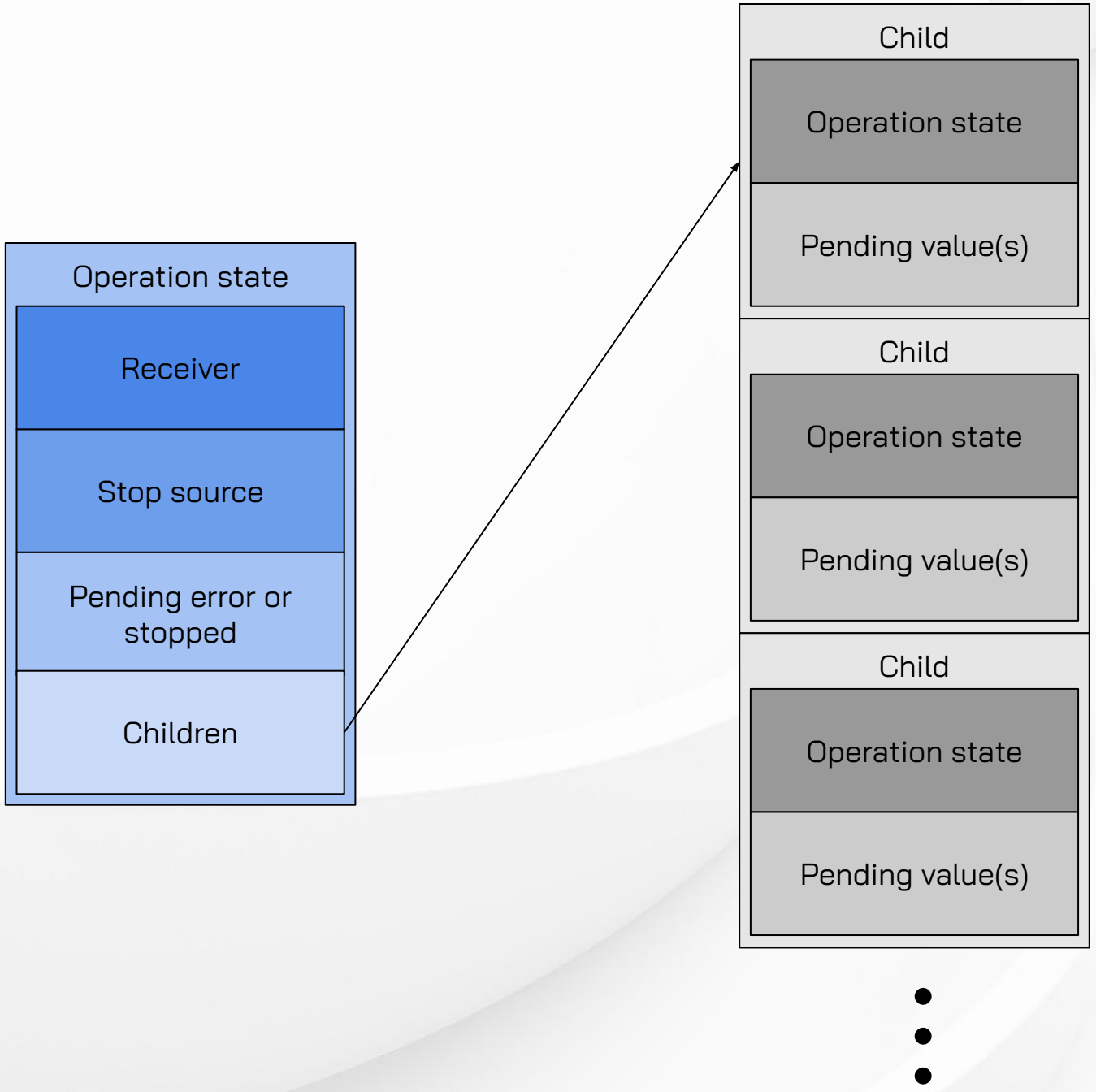
```
template<typename, typename>
inline constexpr bool is_signature_nothrow_visitable_v = true;

template<typename Visitor, typename Tag, typename... Args>
inline constexpr bool is_signature_nothrow_visitable_v<
  Visitor, Tag(Args...)> =
    std::is_nothrow_invocable_v<
      Visitor,
      Tag,
      std::decay_t<Args>...>;
```

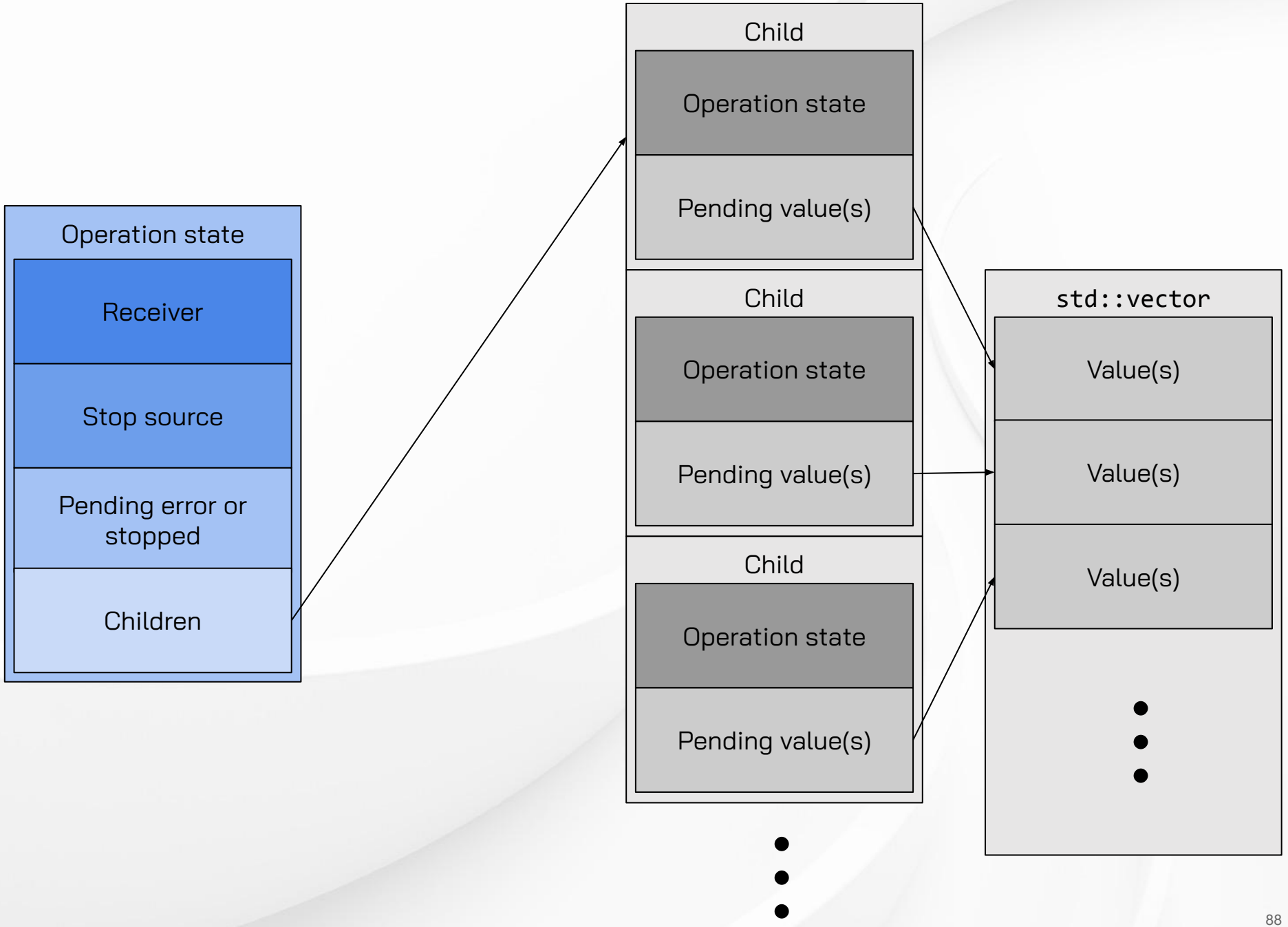
Completion Storage

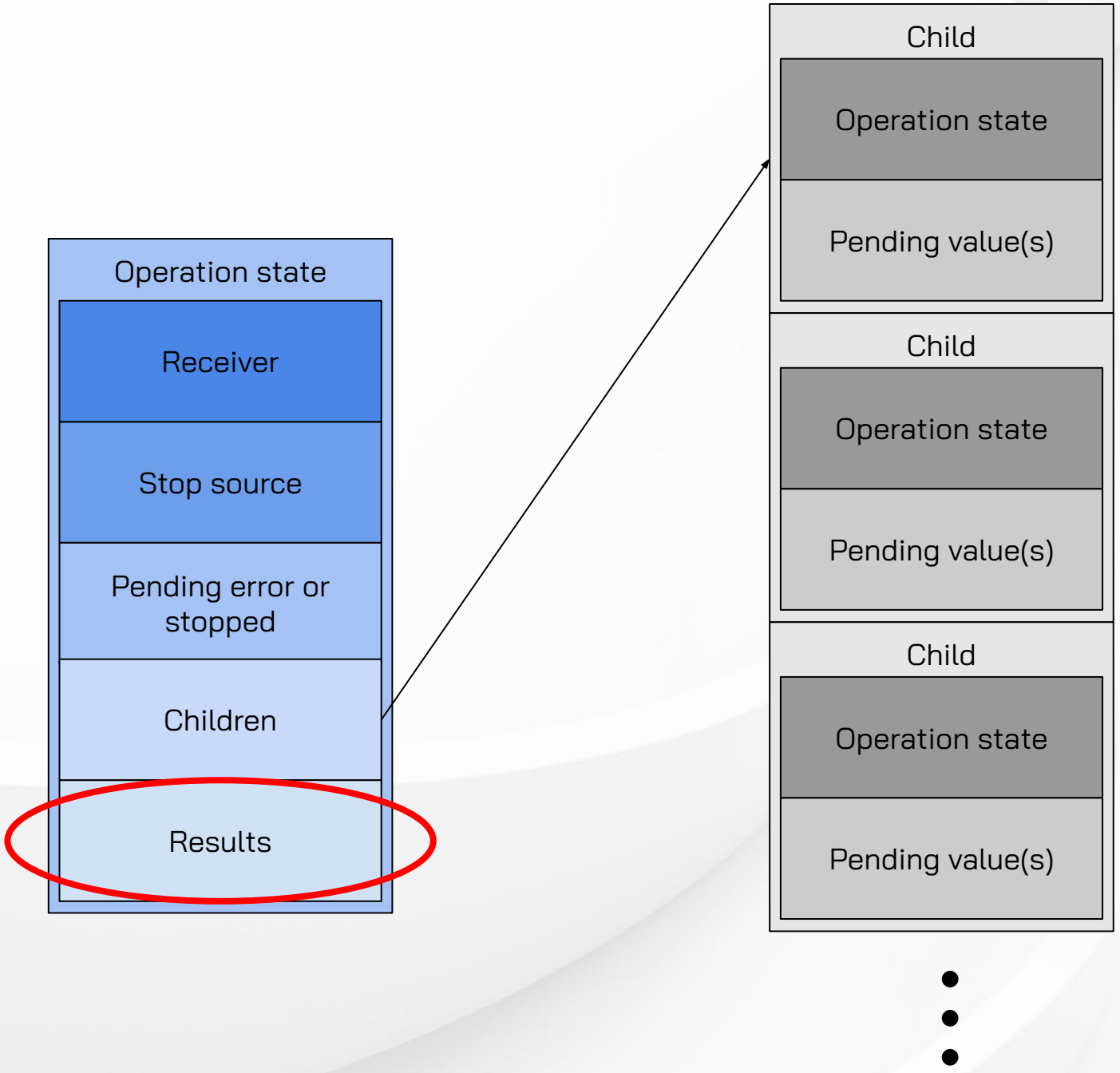
Invokes the visitor with the tag and values.

```
template<typename... Signatures>
class storage_for_completion_signatures</* ... */> {
  // ...
  template<typename Visitor>
  constexpr bool visit(Visitor&& v) && noexcept(
    (is_signature_nothrow_visitable_v<Visitor, Signatures> && ...))
  {
    return std::visit(
      overload(
        [&]<typename... Args>(std::tuple<Args...>&& t) noexcept(
          std::is_nothrow_invocable_v<Visitor, Args...>)
        {
          std::apply(std::forward<Visitor>(v), std::move(t));
          return true;
        },
        [](const std::monostate&) noexcept {
          return false;
        }),
      std::move(storage_));
  }
};
```

LSEG


Operation state
Receiver
Stop source
Pending error or stopped
Children
Child
Operation state
Pending value(s)
Child
Operation state
Pending value(s)
Child
Operation state
Pending value(s)


Operation state
Receiver
Stop source
Pending error or stopped
Children
Child
Operation state
Pending value(s)
Child
Operation state
Pending value(s)
Child
Operation state
Pending value(s)
std::vector
Value(s)
Value(s)
Value(s)


Operation state
Receiver
Stop source
Pending error or stopped
Children
Results
Child
Operation state
Pending value(s)
Child
Operation state
Pending value(s)
Child
Operation state
Pending value(s)

Eric Niebler Today at 2:11 PM

i think of `connect` as the time to acquire resources

Eric Niebler Today at 2:12 PM

`connect` and `start` are separate operations for exactly this reason. so that `start` can (usually) be made no-fail.

Result Storage

`expected` is the number of children and allows the `std::vector`, if any, to have its capacity reserved ahead of time meaning we don't need to allocate as children complete.

`reset` is used to free memory for situations where the overall operation does not succeed.

Invocation delivers the next set of values (i.e. the values sent by the next child).

`complete` sends the final value through to the receiver.

```
struct result {

  explicit result(std::size_t expected, const allocator& alloc);

  void reset() noexcept;

  template<typename... Args>
  void operator()(Args&&...);

  template<typename Receiver>
  void complete(Receiver&&) && noexcept;

};
```

Result Storage

Primary class template is incomplete.

```
template<typename ValueSignatures, typename Env>
struct result;
```

Result Storage

If the children can't complete successfully, there's no need for storage and the operation can't complete with success (hence `std::unreachable`).

```
template<typename Env>
struct result<std::execution::completion_signatures<>, Env> {
  constexpr explicit result(
    const std::size_t,
    const allocator_of_t<Env>&) noexcept
  {}
  static constexpr void reset() noexcept {}
  template<typename Receiver>
  static constexpr void complete(const Receiver&) noexcept {
    std::unreachable();
  }
};
```

Result Storage

If the children send no values, there's no need for a `std::vector`.

```
template<typename Env>
struct result<
  std::execution::completion_signatures<
    std::execution::set_value_t()>,
  Env>
{
  constexpr explicit result(
    const std::size_t,
    const allocator_of_t<Env>&) noexcept
  {}
  static constexpr void reset() noexcept {}
  static constexpr void operator()(
    const std::execution::set_value_t&) noexcept
  {}
  template<typename Receiver>
  static constexpr void complete(Receiver&& r) noexcept {
    std::execution::set_value(std::forward<Receiver>(r));
  }
};
```

LSEG

Result Storage

This specialization actually contains a `std::vector`.

```
template<typename... Args, typename Env>
class result<
  std::execution::completion_signatures<
    std::execution::set_value_t(Args...)>,
  Env>
{
  using storage_type_ = to_vector_t<Env, Args...>;
  using element_ = typename storage_type_::value_type;
  std::optional<storage_type_> storage_;
  /* ...
   *
   *
   *
   */
};
```

LSEG

Result Storage

Constructor reserves which guarantees no allocation will ever be needed at later stages.

```
template<typename... Args, typename Env>
class result</* ... */> {
  // ...
public:
  constexpr explicit result(
    const std::size_t expected,
    const allocator_of_t<Env>& alloc)
    : storage_(
        std::in_place,
        allocator_for_t<element_, Env>(alloc))
  {
    storage_->reserve(expected);
  }
  // ...
};
```

LSEG

Result Storage

Discards all storage and destroys the stored allocator.

```
template<typename... Args, typename Env>
class result</* ... */> {
  // ...
  constexpr void reset() noexcept {
    storage_.reset();
  }
  /* ...
   *
   *
   *
   *
   *
   *
   *
   */
};
```

LSEG

Result Storage

Note that the `noexcept` clause of the function call operator only indicates it can throw if constructing the element can throw, allocation can't throw because we won't reallocate because we reserved sufficient space for all child results.

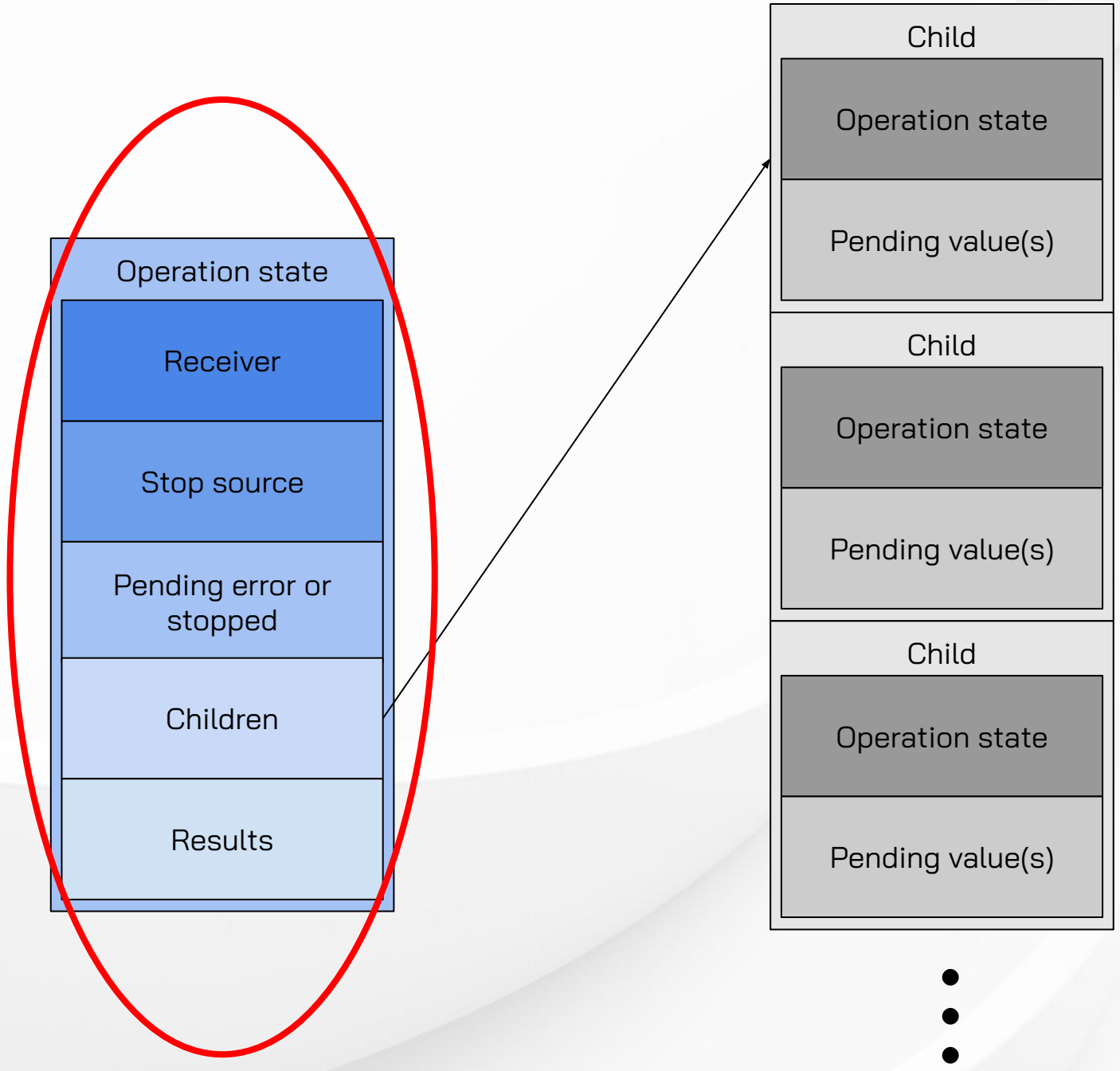
```
template<typename... Args, typename Env>
class result</* ... */> {
  // ...
  template<typename... Ts>
  constexpr void operator()(
    const std::execution::set_value_t&,
    Ts&&... ts) noexcept(
      std::is_nothrow_constructible_v<
        element_,
        Ts...>)
  {
    storage_->emplace_back(std::forward<Ts>(ts)...);
  }
  // ...
};
```

LSEG

Result Storage

Completes the asynchronous operation sending the stored vector of results.

```
template<typename... Args, typename Env>
class result</* ... */> {
  // ...
  template<typename Receiver>
  constexpr void complete(Receiver&& r) && noexcept {
    std::execution::set_value(
      std::forward<Receiver>(r),
      *std::move(storage_));
  }
};
```


Operation state
Receiver
Stop source
Pending error or stopped
Children
Results
Child
Operation state
Pending value(s)
Child
Operation state
Pending value(s)
Child
Operation state
Pending value(s)

Filtering Completion Signatures

Using reflection and ranges this operation is trivial to implement.

```
template<typename Predicate>
consteval auto filter_completion_signatures(
  const std::meta::info signatures,
  Predicate pred)
{
  return substitute(
    ^^std::execution::completion_signatures,
    template_arguments_of(signatures) |
      std::ranges::views::filter(pred));
}
```

LSEG

Extracting Completion Signatures

`value_completion_signatures` extracts the completion signatures which send values.

`other_completion_signatures` extracts the completion signatures which send errors and stopped.

```
template<typename Signatures>
using value_completion_signatures =
  [:filter_completion_signatures(
    ^^Signatures,
    [](std::meta::info info) {
      return return_type_of(info) == ^^std::execution::set_value_t;
    }):];

template<typename Signatures>
using other_completion_signatures =
  [:filter_completion_signatures(
    ^^Signatures,
    [](std::meta::info info) {
      return return_type_of(info) != ^^std::execution::set_value_t;
  }):];
```

LSEG

Sender Type

`decay_t` is applied because we'll end up decay-copying the sender yielded by the range on a later slide.

```
template<typename Range>
using sender_from_range = std::decay_t<
  std::ranges::range_reference_t<Range>>;
```

LSEG

Operation State

Begin with member types and member type aliases.

Note that `state_` is the type which will be used to represent each child.

```
template<typename Range, typename Receiver>
class operation_state : public inlinable_operation_state<
  operation_state<Range, Receiver>, Receiver>
{
  using base_ = inlinable_operation_state<
    operation_state,
    Receiver>;
  using sender_ = sender_from_range<Range>;
  using receiver_env_ = std::env_of_t<Receiver>;
  using env_ = stop_source_env<receiver_env_>;
  using completion_signatures_ =
    std::execution::completion_signatures_of_t<
      sender_,
      env_>;
  using value_completion_signatures_ = value_completion_signatures<
    completion_signatures_>;
  using result_type_ = result<
    value_completion_signatures_,
    receiver_env_>;
  class state_;
  using states_type_ = std::list<
    state_,
    allocator_for_t<state_, receiver_env_>>;
  // ...
};
```

LSEG

Operation State

Member variables.

```
template<typename Range, typename Receiver>
class operation_state : /* ... */ {
  // ...
  chained_stop_source<
    std::execution::stop_token_of_t<receiver_env_>> stop_;
  std::atomic<std::size_t> arrived_{0};
  std::atomic<bool> error_or_stopped_{false};
  [[no_unique_address]]
  storage_for_completion_signatures<
    other_completion_signatures<
      [:completion_signatures(
        dealias(^^completion_signatures_),
        dealias(^^env_)):]>>
          storage_;
  std::optional<states_type_> states_;
  [[no_unique_address]]
  result_type_ result_;
  /* ...
   *
   *
   *
   *
   *
   *
   */
};
```

LSEG

Operation State

Helper private member function for when the operation completes with success.

Note that the children are destroyed before the operation completes.

```
template<typename Range, typename Receiver>
class operation_state : /* ... */ {
  // ...
  void complete_() noexcept {
    states_.reset();
    std::move(result_).complete(std::move(base_::get_receiver()));
  }
  /* ...
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   */
};
```

LSEG

Operation State

Initialization.

```
template<typename Range, typename Receiver>
class operation_state : /* ... */ {
  // ...
public:
  constexpr explicit operation_state(Range&& r, Receiver rcvr)
    : base_(std::move(rcvr)),
      stop_(
        std::execution::get_stop_token(
          std::get_env(base_::get_receiver()))),
      states_(/* ... */),
      result_(/* ... */)
  {}
  /* ...
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   */
};
```

LSEG

Operation State

Creation of a state for each child.

As we'll see in later slides this connects the child sender to a receiver to form an operation state.

```
template<typename Range, typename Receiver>
class operation_state : /* ... */ {
  // ...
public:
  constexpr explicit operation_state(Range&& r, Receiver rcvr)
    : base_(/* ... */),
      stop_(/* ... */),
      states_([&]() {
        states_type_ retr(
          ::get_allocator_for<state_>(
            std::get_env(base_::get_receiver())));
        for (auto&& sender : r) {
          retr.emplace_back(
            *this,
            std::forward<decltype(sender)>(sender));
        }
        return retr;
      }()),
      result_(/* ... */)
  {}
  /* ...
   *
   *
   */
};
```

LSEG

Operation State

Initialization.

```
template<typename Range, typename Receiver>
class operation_state : /* ... */ {
  // ...
public:
  constexpr explicit operation_state(Range&& r, Receiver rcvr)
    : base_(/* ... */),
      stop_(/* ... */),
      states_(/* ... */),
      result_(
        states_->size(),
        ::get_allocator(std::get_env(base_::get_receiver())))
  {}
  /* ...
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   */
};
```

LSEG

Operation State

If there are no children starting the operation completes immediately.

Otherwise starting the operation loops over each child and starts it.

But there's a subtle problem with this code...

```
template<typename Range, typename Receiver>
class operation_state : /* ... */ {
  // ...
  constexpr void start() & noexcept {
    if (states_.empty()) {
      complete_();
      return;
    }
    for (auto&& state : states_) {
      state.start();
    }
  }
  /* ...
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   */
};
```

LSEG

Operation State

The last child may complete synchronously and lead to the end of the lifetime of `*this` and therefore we need to be careful not to access or use any members of ourselves after starting the last child.

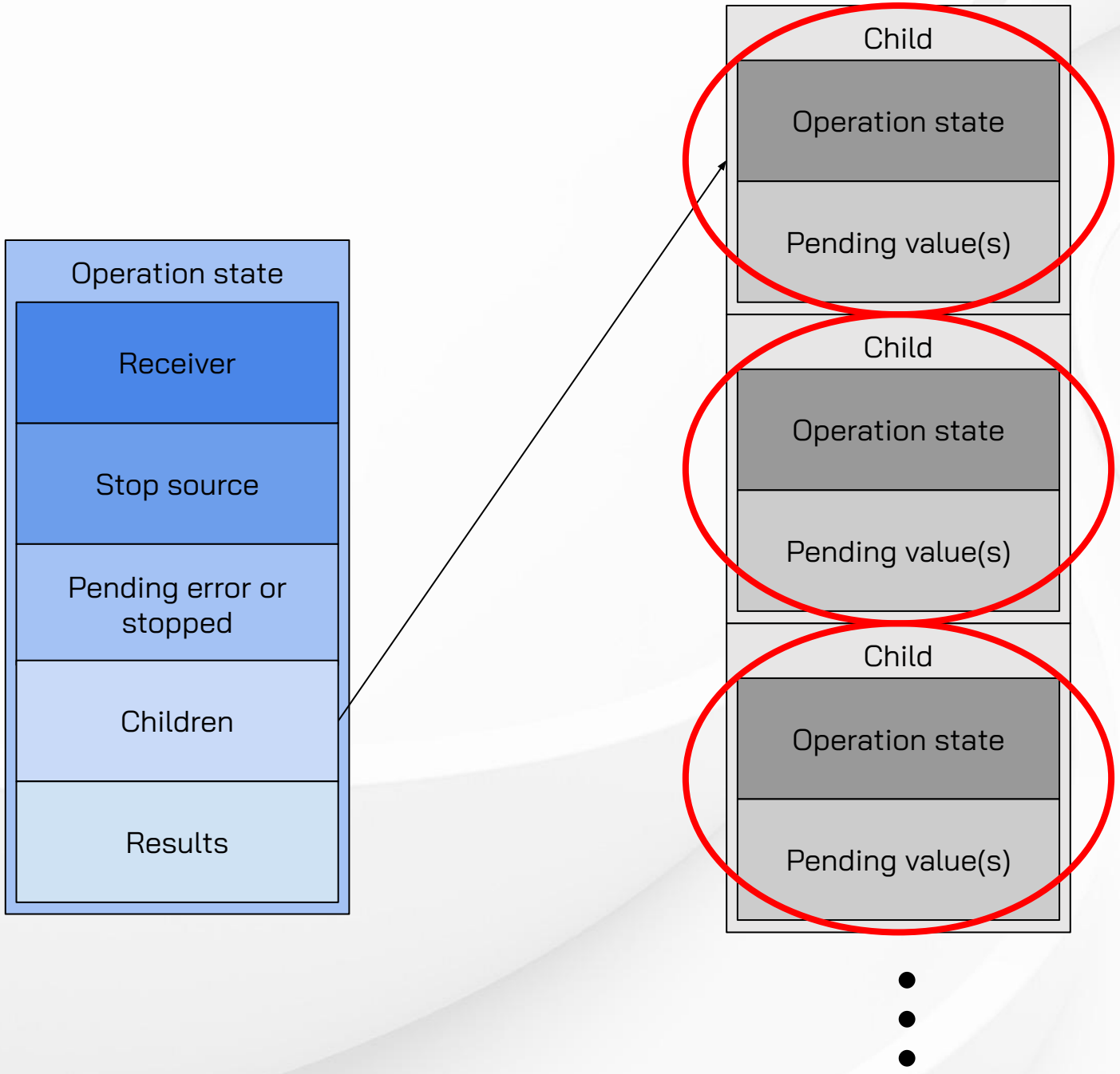
```
template<typename Range, typename Receiver>
class operation_state : /* ... */ {
  // ...
  constexpr void start() & noexcept {
    auto iter = states_->begin();
    if (iter == states_->end()) {
      complete_();
      return;
    }
    for (;;) {
      auto&& state = *iter;
      ++iter;
      const auto last = iter == states_->end();
      state.start();
      if (last) {
        break;
      }
    }
  }
  /* ...
   *
   *
   *
   */
};
```

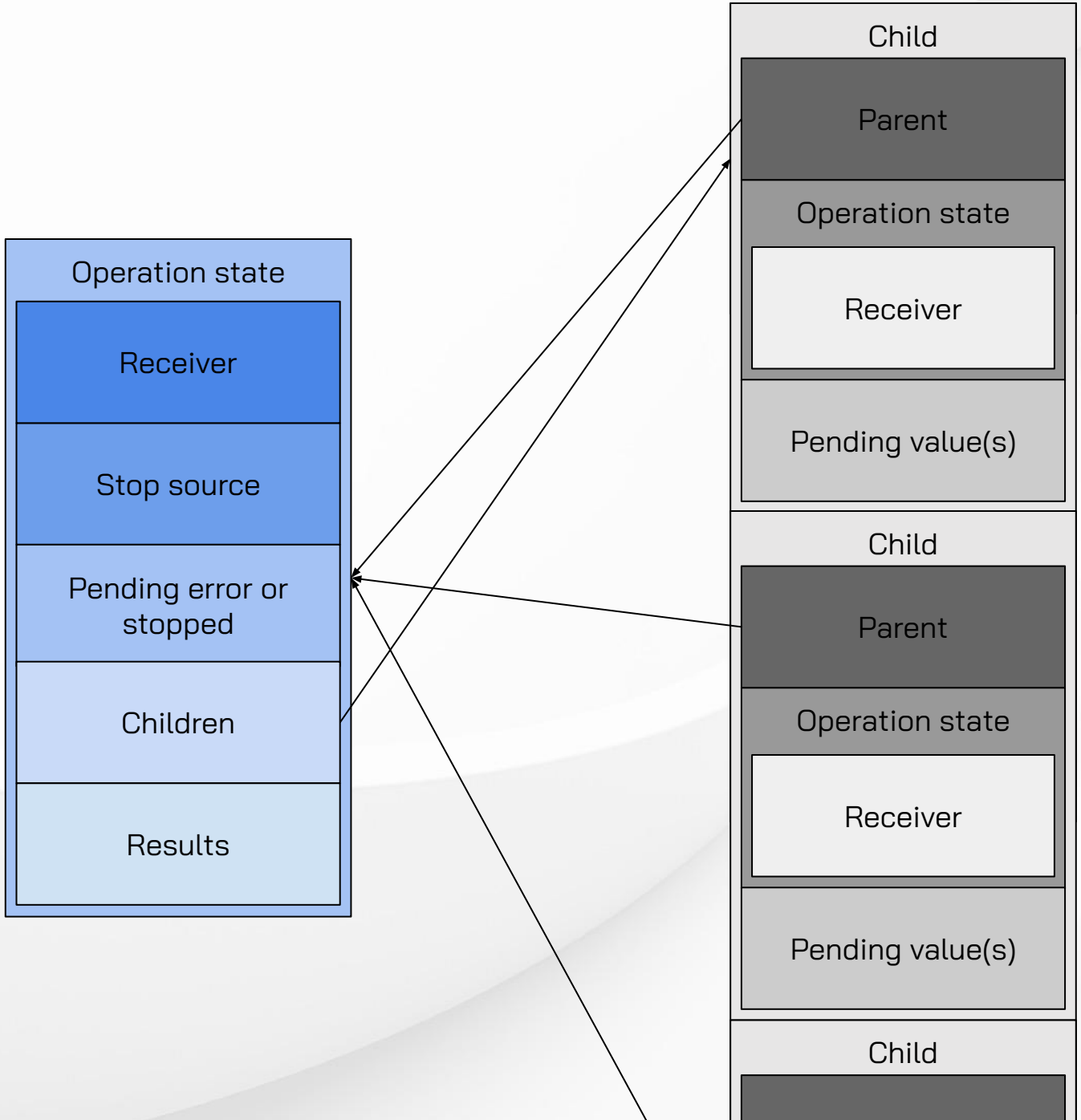
Operation State

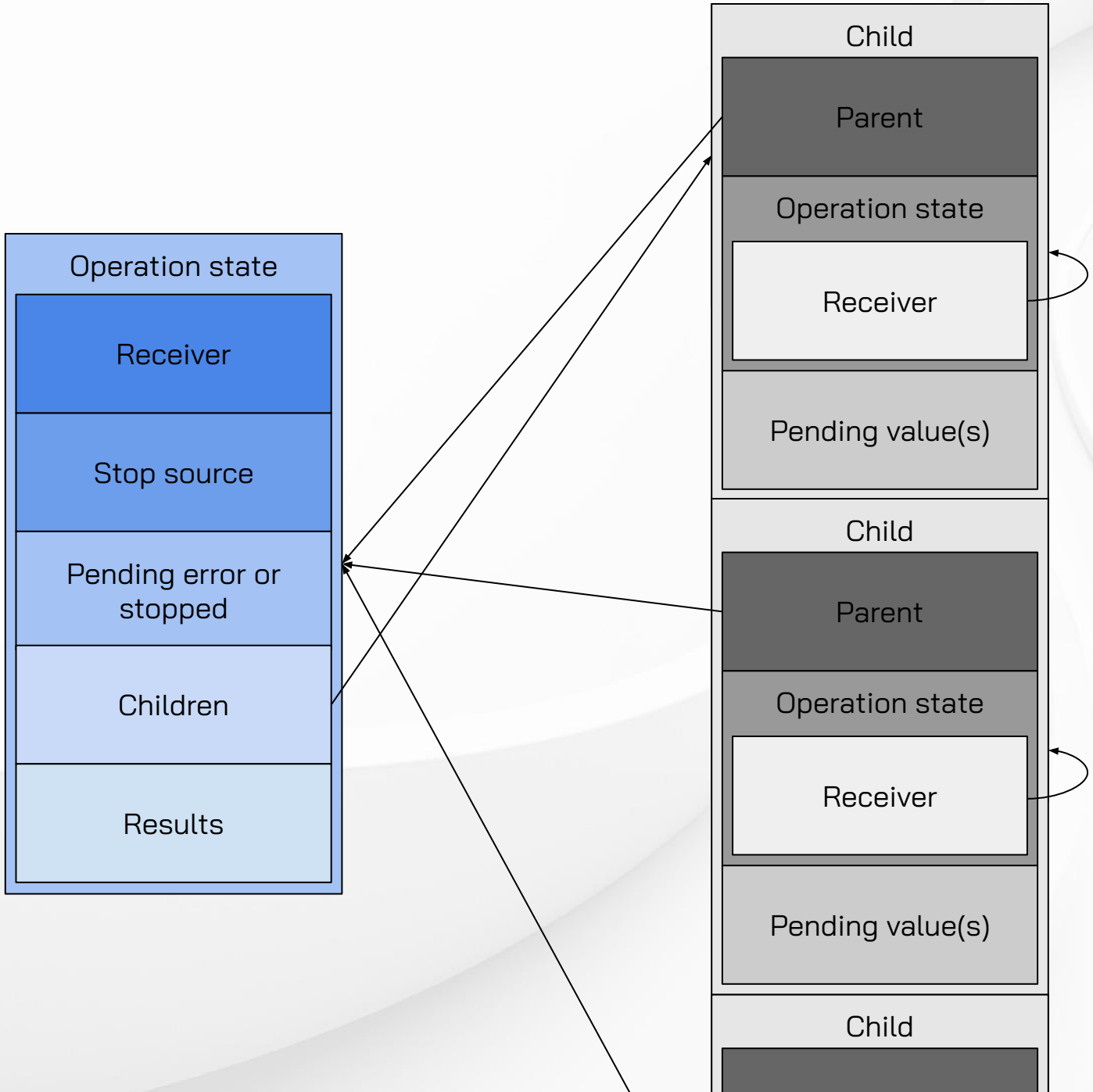
Don’t forget the opt-in concept!

```
template<typename Range, typename Receiver>
class operation_state : /* ... */ {
  // ...
  using operation_state_concept =
    std::execution::operation_state_t;
  /* ...
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   */
};
```

LSEG


Operation state
Receiver
Stop source
Pending error or stopped
Children
Results
Child
Operation state
Pending value(s)
Child
Operation state
Pending value(s)
Child
Operation state
Pending value(s)


Operation state
Receiver
Stop source
Pending error or stopped
Children
Results
Child
Parent
Operation state
Receiver
Pending value(s)
Child
Parent
Operation state
Receiver
Pending value(s)
Child


Operation state
Receiver
Stop source
Pending error or stopped
Children
Results
Child
Parent
Operation state
Receiver
Pending value(s)
Child
Parent
Operation state
Receiver
Pending value(s)
Child

D3425R1: Reducing operation-state sizes for subobject child operations

Table of Contents

P3425

CRTP base class for operation states for operations that want to pass child operations an inlinable receiver.

```
template<typename ParentOp, typename Env, typename ChildSender>
class child_operation_state {
  class receiver_ {
    ParentOp* parent_op_;
  public:
    using receiver_concept = std::execution::receiver_t;
    constexpr explicit receiver_(ParentOp* parent_op) noexcept
      : parent_op_(parent_op)
    {}
    template<typename ChildOp>
    constexpr static receiver_ make_receiver_for(ChildOp* child_op)
      noexcept
    {
      return receiver_(/* ... */);
    }
    /* ...
     *
     *
     */
  };
  // ...
};
```

LSEG

It is not permitted to go from the address of a child data-member to the address of the parent class except in very limited circumstances. This rule is there to permit, among other things, a compiler-optimisation called "scalar replacement of aggregates", which allows the compiler to break up an aggregate type into a set of separate stack-allocations for each of the data-members if the address of the parent object is not aliased/observed.

The very limited circumstances in which we can go from the address of a sub-object to the address of the parent-object are the following:

- When the sub-object is a non-ambiguous base-class of parent-object ([expr.static.cast] p11) In this case, we can use `static_cast` to cast from pointer to base-class to the pointer to the derived parent-object
- When the parent-object and sub-object are "pointer-interconvertible" ([basic.compound] p5). In this case, we can use `reinterpret_cast` to cast from pointer to sub-object to pointer to parent-object.

Two objects are "pointer-interconvertible" only if:

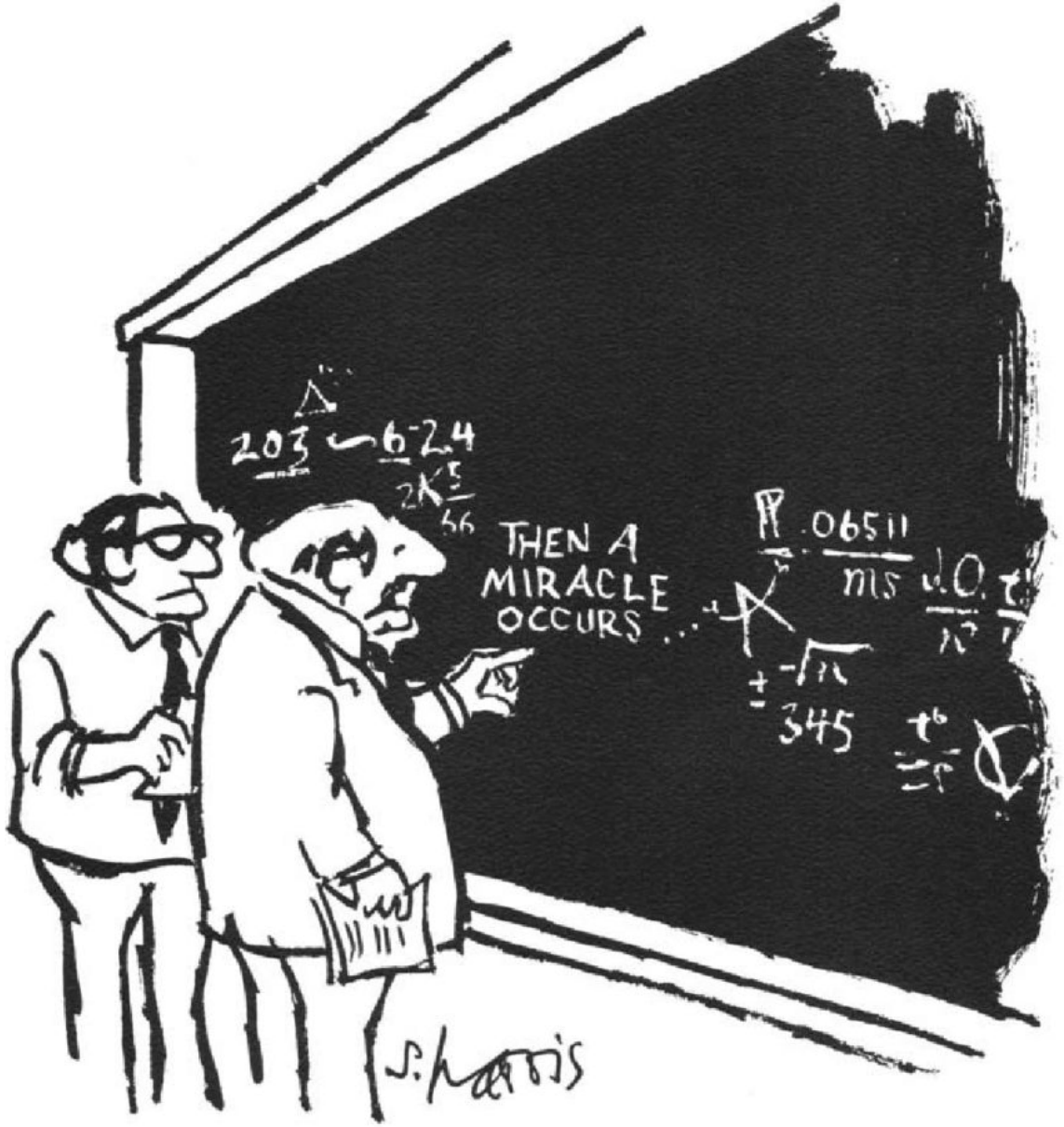
- the parent-object is a union and the sub-object is a non-static data-member of that union; or
- the parent-object is a "standard layout" class object and the sub-object is the first non-static data-member of the parent-object or any base-class sub-object of the parent-object
- there exists an intermediate sub-object, `C`, such that the parent-object is pointer-interconvertible with `C` and `C` is pointer-interconvertible with the sub-object (i.e. the relationship is transitive)

Note that there are a number of rules for types that are considered "standard layout" class types ([class.prop] p3). I won't go into particular details here but, among other things, this doesn't allow types with virtual methods, virtual base-classes, types with non-static data-members with different access control, or data-members that are not also standard layout class types.

As child operation states in general are not going to all be standard layout types and since we also want to support cases where a parent-operation has multiple child operations, we cannot just rely on being able to convert the address of the first non-static data member to the address of the parent as a general solution.

This means that we are going to need to make use of base-classes to allow going from address of a sub-object to the address of a parent-object.

Further, there are also cases where we need to be able to defer construction of a child operation-state until after the operation is started, or where we



"I THINK YOU SHOULD BE MORE EXPLICIT HERE IN STEP TWO."

P3425

The inlinable receiver just forwards all calls through to the parent operation state.

```
template<typename ParentOp, typename Env, typename ChildSender>
class child_operation_state {
  class receiver_ {
    // ...
    template<typename... Args>
    constexpr void set_value(Args&&... args) && noexcept {
      parent_op_->set_value(std::forward<Args>(args)...);
    }
    template<typename... Args>
    constexpr void set_error(Args&&... args) && noexcept {
      parent_op_->set_error(std::forward<Args>(args)...);
    }
    template<typename... Args>
    constexpr void set_stopped(Args&&... args) && noexcept {
      parent_op_->set_stopped(std::forward<Args>(args)...);
    }
    constexpr Env get_env() const noexcept {
      return parent_op_->get_env();
    }
  };
  // ...
};
```

LSEG

P3425

A lot of missing implementation on this slide.

```
template<typename ParentOp, typename Env, typename ChildSender>
class child_operation_state {
  // ...
protected:
  constexpr child_operation_state(ChildSender&& sender) noexcept(
    noexcept(
      std::execution::connect(
        std::forward<ChildSender>(sender),
        std::declval<receiver_>())))
  {
    /* ... */
  }
  constexpr void start() & noexcept {
    std::execution::start(/* ... */);
  }
};
```

LSEG

Operation State

Final entries in our operation state to support implementation of the child state.

```
template<typename Range, typename Receiver>
class operation_state : /* ... */ {
  // ...
private:
  using child_operation_state_ = child_operation_state<
    state_,
    env_,
    sender_>;
  class state_base_ {
  protected:
    operation_state& self_;
    constexpr explicit state_base_(operation_state& self) noexcept
      : self_(self)
    {}
  };
};
```

LSEG

Child State

Storage for the completion which will be generated by the child operation.

```
class operation_state<Range, Receiver>::state_
  : state_base_, public child_operation_state_
{
  [[no_unique_address]]
  storage_for_completion_signatures<
    value_completion_signatures_> storage_;
  using state_base_::self_;
  /* ...
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   */
};
```

LSEG

Child State

`maybe_complete_` is the meat of `state_`: It checks to see if the overall operation has been completed by the arrival of this child and, if so, completes it.

```
class operation_state<Range, Receiver>::state_ : /* ... */ {
  // ...
  constexpr void maybe_complete_() noexcept {
    auto&& op = self_;
    const auto error = [&]() noexcept {
      return std::move(op.storage_).visit(
        [&](const auto& tag, auto&&... args) noexcept {
          op.states_.reset();
          op.result_.reset();
          tag(
            std::move(op.get_receiver()),
            std::forward<decltype(args)>(args)...);
        });
    };
    /* ...
     *
     *
     *
     *
     *
     */
  }
  // ...
};
```

LSEG

Child State

These two lines are incredibly important to consider together.

The declaration of `op` doesn't just make referring to the parent (`when_all_range`) operation state more terse, it also ensures that reference has automatic storage duration.

This is important because `op.states_.reset()` has the effect of ending the lifetime of `*this`, thereafter it's unsafe to access any member of ourselves meaning we need op in order to continue accessing the parent.

```
class operation_state<Range, Receiver>::state_ : /* ... */ {
  // ...
  constexpr void maybe_complete_() noexcept {
    auto&& op = self_;
    const auto error = [&]() noexcept {
      return std::move(op.storage_).visit(
        [&](const auto& tag, auto&&... args) noexcept {
          op.states_.reset();
          op.result_.reset();
          tag(
            std::move(op.get_receiver()),
            std::forward<decltype(args)>(args)...);
        });
    };
    /* ...
     *
     *
     *
     *
     *
     */
  }
  // ...
};
```

Child State

The atomic increment on this slide is the only synchronization anywhere in this operation.

Note that if all children haven't completed, we bail out, and if we end of completing in error, we bail out (the receiver contract is satisfied).

```
class operation_state<Range, Receiver>::state_ : /* ... */ {
  // ...
  constexpr void maybe_complete_() noexcept {
    // ...
    const auto children = op.states_->size();
    const auto arrived =
      op.arrived_.fetch_add(1, std::memory_order_acq_rel) + 1U;
    if (arrived != children) {
      return;
    }
    if (error()) {
      return;
    }
    /* ...
     *
     *
     *
     *
     *
     *
     *
     */
  }
  // ...
};
```

LSEG

Child State

The loop collects results from each child. Note that this is accomplished by visiting the child's storage into the result storage of the parent operation state.

```
class operation_state<Range, Receiver>::state_ : /* ... */ {
  // ...
  constexpr void maybe_complete_() noexcept {
    // ...
    for (auto&& state : *op.states_) {
      constexpr auto nothrow = noexcept(
        std::move(state.storage_).visit(op.result_));
      const auto impl = [&]() noexcept(nothrow) {
        if (!std::move(state.storage_).visit(op.result_)) {
          std::unreachable();
        }
      };
      /* ...
       *
       *
       *
       *
       *
       */
    }
    // ...
  }
  // ...
};
```

LSEG

Child State

Boilerplate to deal with the possibility of exceptions.

Once loop completes overall operation (`when_all_range`) completes.

```
class operation_state<Range, Receiver>::state_ : /* ... */ {
  // ...
  constexpr void maybe_complete_() noexcept {
    // ...
    for (auto&& state : *op.states_) {
      // ...
      if constexpr (nothrow) {
        impl();
      } else {
        try {
          impl();
        } catch (...) {
          op.storage_.arrive(
            std::execution::set_error_t{},
            std::current_exception());
          if (!error()) std::unreachable();
          return;
        }
      }
    }
    op.complete_();
  }
  // ...
};
```

LSEG

Child State

We only record one error or one instance of stopped per parent operation, this implements that logic.

Note that since relaxed memory ordering is used there's no synchronization here because there's nothing to synchronize. We don't care about happens-before relationships we just care that storage occurs once.

```
class operation_state<Range, Receiver>::state_ : /* ... */ {
  // ...
  template<typename... Args>
  constexpr void error_or_stopped_(Args&&... args) noexcept {
    if (!self_.error_or_stopped_.exchange(
      true,
      std::memory_order_relaxed))
    {
      self_.stop_.request_stop();
      constexpr auto nothrow = noexcept(
        self_.storage_.arrive(std::forward<Args>(args)...));
      const auto impl = [&]() noexcept(nothrow) {
        self_.storage_.arrive(std::forward<Args>(args)...);
      };
      /* ...
       *
       *
       *
       */
    }
    // ...
  }
  // ...
};
```

LSEG

Child State

Just because it's an error doesn't mean storing it can't throw, so we handle that.

```
class operation_state<Range, Receiver>::state_ : /* ... */ {
  // ...
  template<typename... Args>
  constexpr void error_or_stopped_(Args&&... args) noexcept {
    if (/* ... */) {
      // ...
      if constexpr (nothrow) {
        impl();
      } else {
        try {
          impl();
        } catch (...) {
          self_.storage_.arrive(
            std::execution::set_error_t{},
            std::current_exception());
        }
      }
    }
    maybe_complete_();
  }
  /* ...
   *
   */
};
```

LSEG

Child State

Environment retrieval.

```
class operation_state<Range, Receiver>::state_ : /* ... */ {
  // ...
public:
  using child_operation_state_::start;
  constexpr env_ get_env() const noexcept {
    return env_(
      self_.stop_,
      std::get_env(self_.get_receiver()));
  }
  /* ...
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   */
};
```

LSEG

Child State

Constructor and boilerplate.

Note that `state_base_` is initialized before `child_operation_state_`. This is vitally important because constructing `child_operation_state_` might obtain the environment, which requires the reference back to the parent be initialized.

```
class operation_state<Range, Receiver>::state_ : /* ... */ {
  // ...
  constexpr explicit state_(
    operation_state& self,
    sender_ s) noexcept(
      std::is_nothrow_constructible_v<
        child_operation_state_,
        sender_>)
    : state_base_(self),
      child_operation_state_(std::move(s))
  {}
  /* ...
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   */
};
```

LSEG

Child State

Handling value completion signal.

```
class operation_state<Range, Receiver>::state_ : /* ... */ {
  // ...
  template<typename... Args>
  constexpr void set_value(Args&&... args) noexcept {
    constexpr auto nothrow =
      (std::is_nothrow_constructible_v<
        std::decay_t<Args>,
        Args> && ...);
    const auto impl = [&]() noexcept(nothrow) {
      storage_.arrive(
        std::execution::set_value_t{},
        std::forward<Args>(args)...);
    };
    /* ...
     *
     *
     *
     *
     *
     *
     *
     */
  }
  // ...
};
```

LSEG

Child State

Deal with exceptions and then complete if this is the last child to complete.

```
class operation_state<Range, Receiver>::state_ : /* ... */ {
  // ...
  template<typename... Args>
  constexpr void set_value(Args&&... args) noexcept {
    // ...
    if constexpr (nothrow) {
      impl();
    } else {
      try {
        impl();
      } catch (...) {
        error_or_stopped_(
          std::execution::set_error_t{},
          std::current_exception());
        return;
      }
    }
    maybe_complete_();
  }
  /* ...
   *
   *
   */
};
```

LSEG

Child State

Handling error and stopped completion signals is exactly the same except the tag.

```
class operation_state<Range, Receiver>::state_ : /* ... */ {
  // ...
  template<typename... Args>
  constexpr void set_error(Args&&... args) noexcept {
    error_or_stopped_(
      std::execution::set_error_t{},
      std::forward<Args>(args)...);
  }
  template<typename... Args>
  constexpr void set_stopped(Args&&... args) noexcept {
    error_or_stopped_(
      std::execution::set_stopped_t{},
      std::forward<Args>(args)...);
  }
};
```

Sender

Storage for the range, opting into the `std::execution::sender` concept, and the constructor.

```
template<typename Range>
class sender {
  Range r_;
public:
  using sender_concept = std::execution::sender_t;
  template<typename T>
    requires std::constructible_from<Range, T>
  constexpr explicit sender(T&& t) noexcept(
    std::is_nothrow_constructible_v<Range, T>)
    : r_(std::forward<T>(t))
  {}
  /* ...
   *
   *
   *
   */
};
```

LSEG

Sender

Helper type computations.

`range_` yields Range except with cv- and ref-qualifications copied from Self.

`completion_signatures_` yields the completion signatures of the child sender.

```
template<typename Range>
class sender {
  // ...
private:
  template<typename Self>
  using range_ =
    decltype(std::forward_like<Self>(std::declval<Range&>()));
  template<typename Self, typename Env>
  using completion_signatures_ =
    std::execution::completion_signatures_of_t<
      sender_from_range<range_<Self>>,
      stop_source_env<Env>>;
  /* ...
   *
   *
   */
};
```

LSEG


Shottedness

Shottedness

With what qualifications can the sender be connected?

- An invocable may be:
  - Rvalue invocable, in which case it may only be invoked once
  - Lvalue invocable, in which case it may be invoked many times
- Isomorphically senders may be:
  - “Single shot,” if they can only be rvalue connected
  - “Multishot,” if they can be lvalue connected

Ranges & Shottedness

How do range qualities and properties map onto shottedness?

- Connecting a `when_all_range` constructs an operation state which traverses the range wrapped by the sender
- Input ranges do not offer the multi-pass guarantee, and therefore can only be traversed once
- Ranges stronger than input do offer the multi-pass guarantee, and therefore can be traversed multiple times

Range Category	**Shottedness**
Input	Single
Forward or stronger	Multiple

Sender

We only generate completion signatures if we can be connected with the given qualification.

If all range considerations are satisfied we yield the completion signatures we computed previously.

Note that `completion_signatures` might fail to be a constant expression in which case the sender has no completion signatures and will not be connectable (see next slide).

```
template<typename Range>
class sender {
  // ...
public:
  template<typename Self, typename Env>
    requires (
      !std::is_lvalue_reference_v<range_<Self>> ||
      std::ranges::forward_range<range_<Self>>)
  consteval static auto get_completion_signatures() noexcept ->
    [:completion_signatures(
      dealias(^^completion_signatures_<Self, Env>),
      ^^stop_source_env<Env>):]
  {
    return {};
  }
  // ...
};
```

Sender

Connects sender and receiver to form an operation state.

Note that since the constraint depends on `get_completion_signatures` (previous slide) the sender is not connectable if that's ill-formed.

```
template<typename Range>
class sender {
  // ...
  template<typename Self, typename Receiver>
    requires std::execution::receiver_of<
      Receiver,
      std::execution::completion_signatures_of_t<
        Self,
        std::env_of_t<Receiver>>>
  auto connect(this Self&& self, Receiver receiver) {
    return operation_state<range_<Self>, Receiver>(
      std::forward<Self>(self).r_,
      std::move(receiver));
  }
};
```

LSEG

Sender Factory

From the beginning of the talk.

```
template<std::ranges::input_range Range>
  requires
    std::execution::sender<
      std::ranges::range_reference_t<Range>> &&
    std::constructible_from<std::decay_t<Range>, Range>
std::execution::sender auto when_all_range(Range&& range);
```

LSEG

Sender Factory

Now complete.

```
template<std::ranges::input_range Range>
  requires
    std::execution::sender<
      std::ranges::range_reference_t<Range>> &&
    std::constructible_from<std::decay_t<Range>, Range>
std::execution::sender auto when_all_range(Range&& range)
  noexcept(
    std::is_nothrow_constructible_v<
      sender<std::decay_t<Range>>,
      Range>)
{
  return sender<std::decay_t<Range>>(std::forward<Range>(range));
}
```

LSEG

Why did we do all of this?

when_all_range Example

Example from the beginning of the talk had a fixed cardinality of input sender (2), this allows any number of children to be passed in by varying `upper_bound`.

```
const unsigned upper_bound = 6;
timed_thread_context ctx;
std::this_thread::sync_wait(
  when_all_range(
    std::ranges::views::iota(1U, upper_bound) |
    std::ranges::views::transform([&](const unsigned u) {
      return
        schedule_after(
          ctx.get_scheduler(),
          std::chrono::seconds(u)) |
        std::execution::then([u]() {
          char c('a' - 1);
          c += u;
          std::cout << c << std::endl;
        });
    })));
```

LSEG

a

b

c

d

e

duration 5000 ms

Summary

Generic Asynchronous Programming

- Reflection enables computation in the type domain (including completion signatures) without pains of “regular” template metaprogramming
- `std::execution` provides a general framework for expressing asynchronous computation
- Generic programming still involves many edge cases and subtleties
- `std::execution` algorithms provide the genericity and reusability expected of C++, executed asynchronously

Thank you

Robert Leahy
Lead Software Engineer
rleahy@rleahy.ca

LSEG

Questions?

LSEG