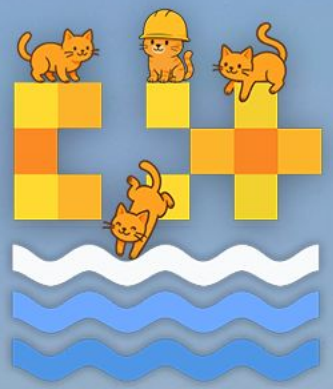


Extending `std::execution` Further

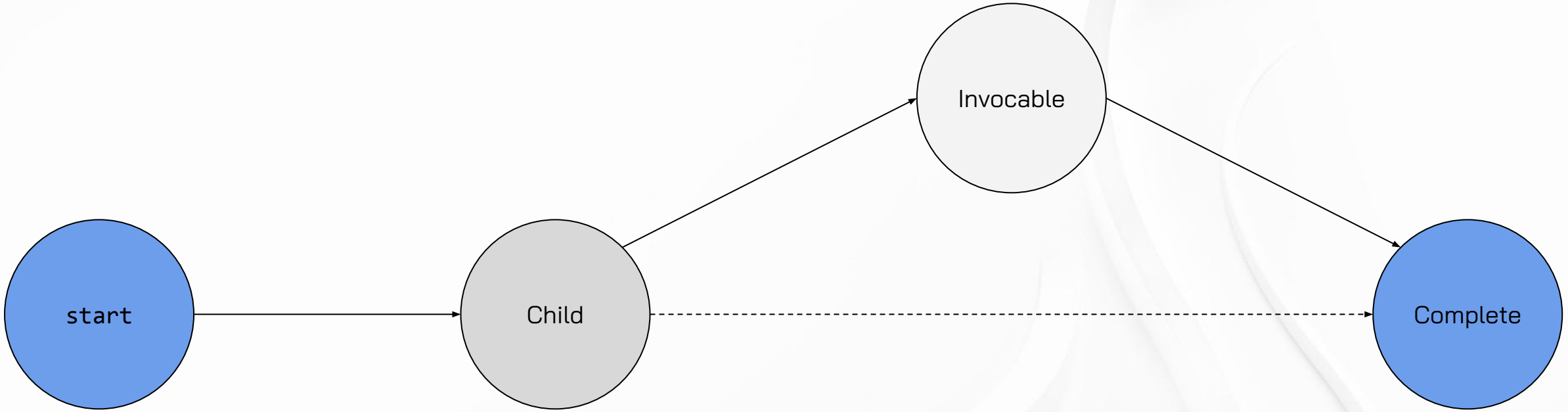
Higher-Order Senders and the Shape of Asynchronous Programs

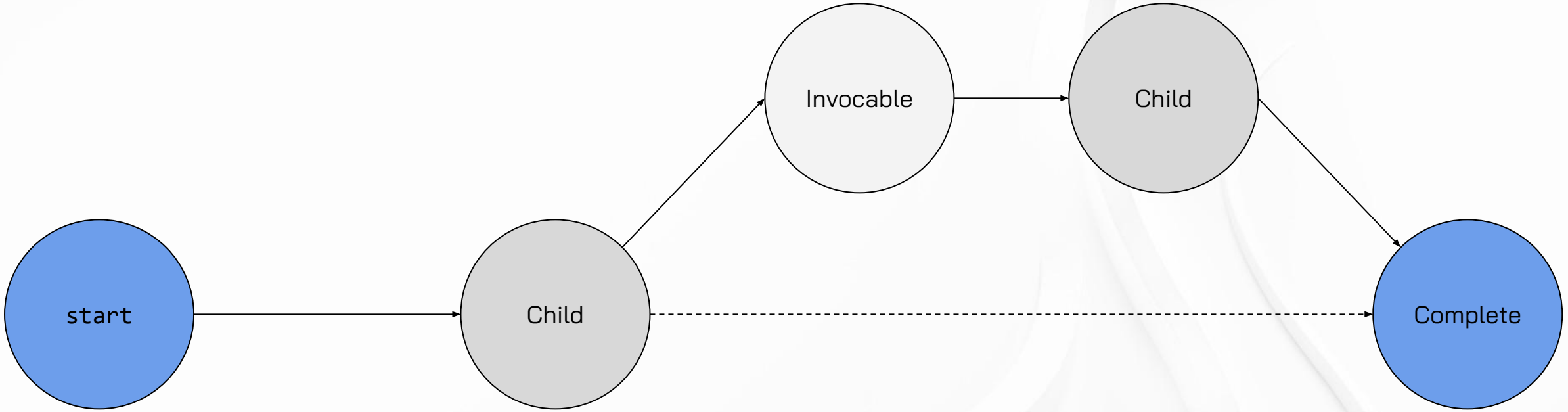
Robert Leahy

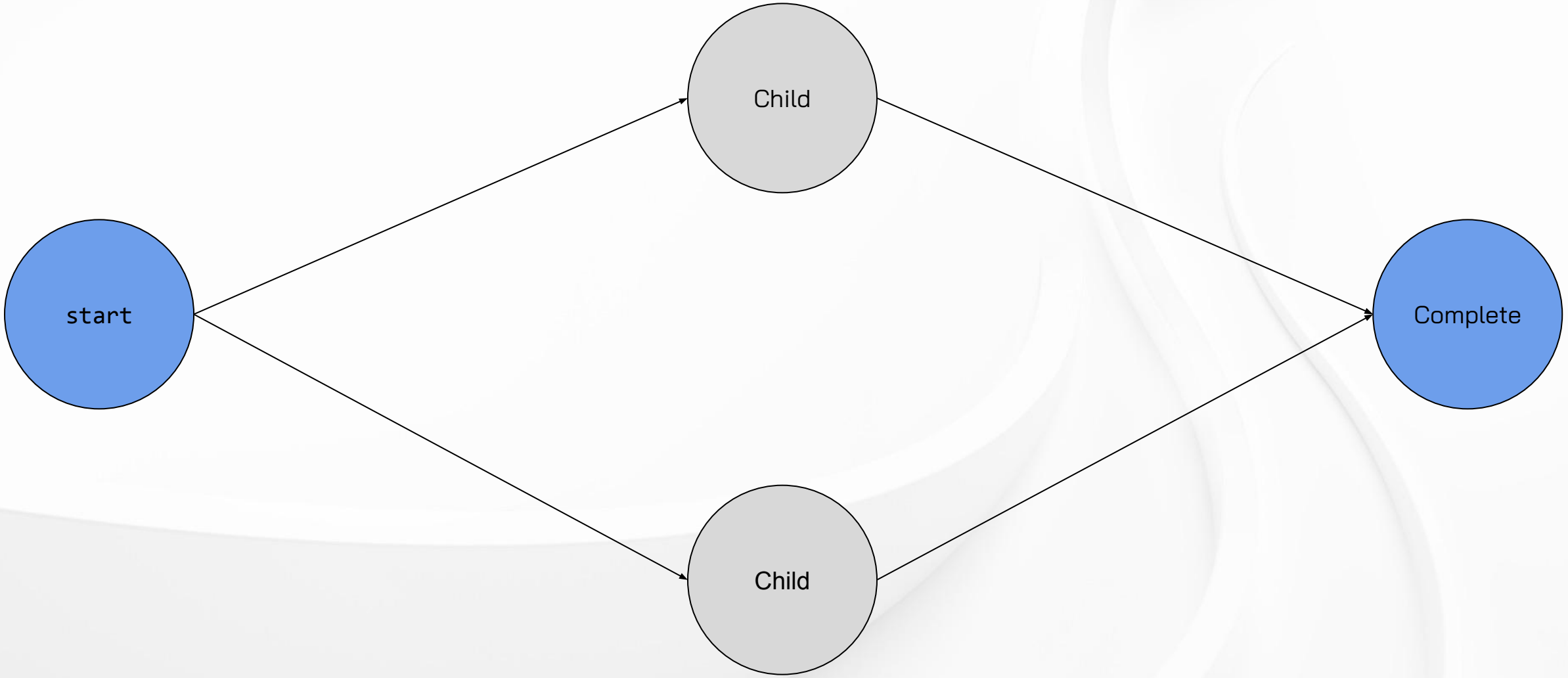
2025

Let’s review


start
Complete


start
Child
Invocable
Complete


start
Child
Invocable
Child
Complete


start
Child
Child
Complete

Do real programs look like this?

LSEG

Network Server

The fact that this is the core loop of a network server isn't what's relevant.

What's relevant is the structure, common to many applications: Waiting for an event, handling that event, and then repeating.

```
void handle_connection(int fd);

const int accepting = /* ... */;
for (;;) {
  const auto conn = accept(accepting, nullptr, nullptr);
  if (conn == -1) {
    throw std::system_error(/* ... */);
  }
  handle_connection(conn);
}
```

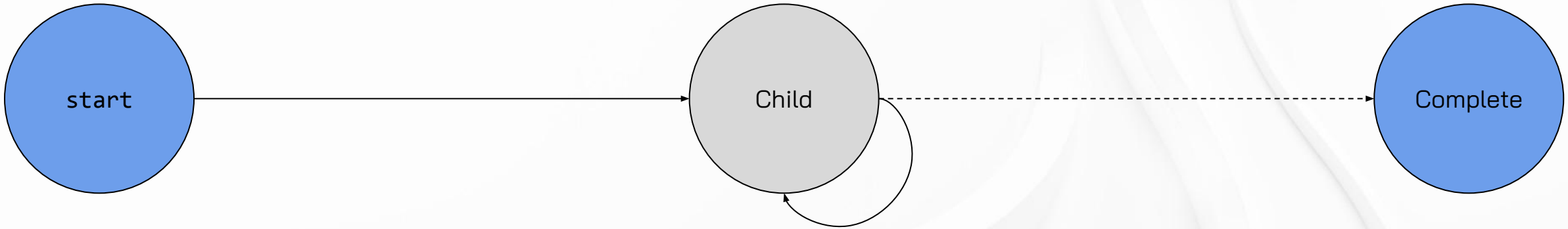

Core C++ 2024
Evolving C++ Networking With Senders & Receivers
Part 1 & 2
Robert Leahy

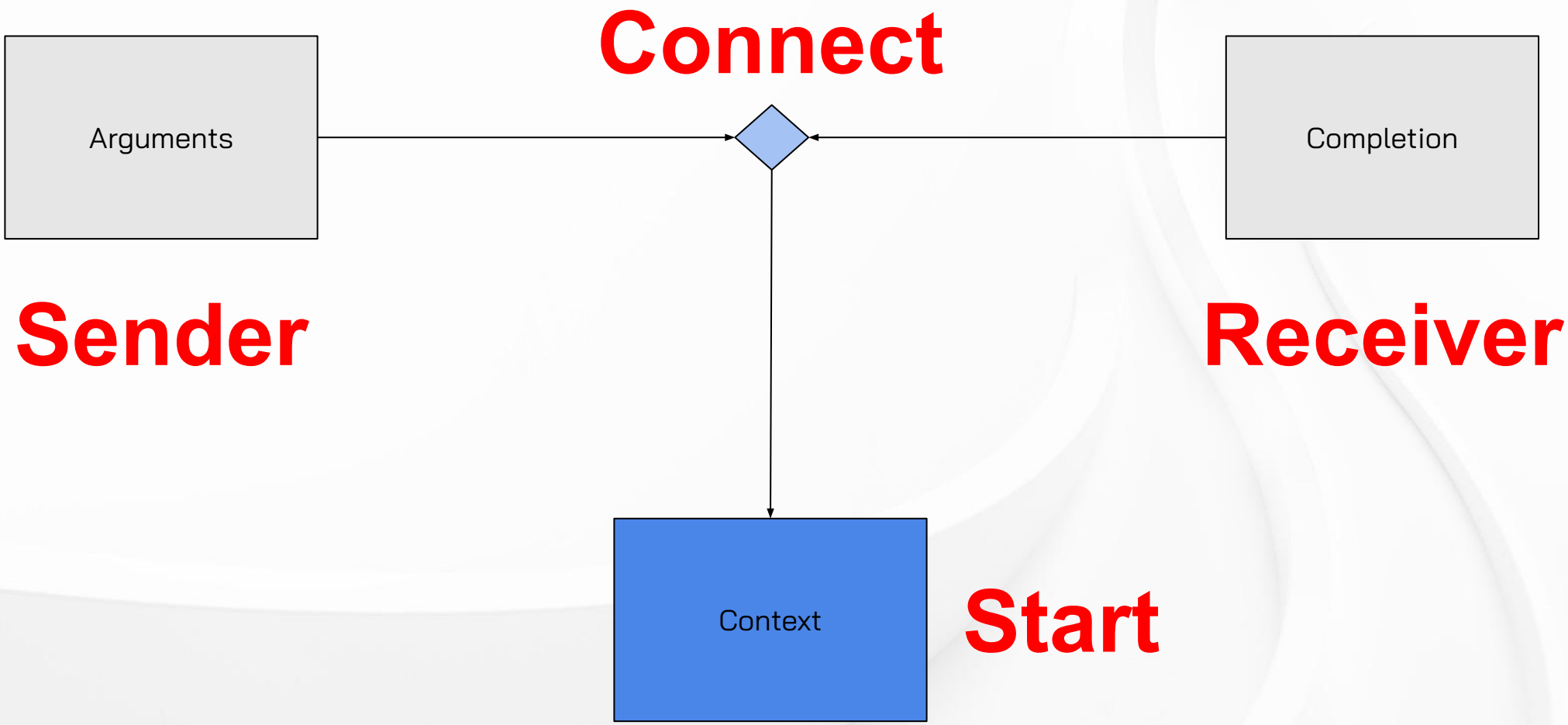
repeat

Repeatedly connects and starts a certain sender.

```
struct repeat_t : std::execution::sender_adaptor_closure<repeat_t>
{

  template<std::execution::sender Sender>
  static constexpr std::execution::sender auto operator()(
    Sender&& sender) noexcept(/* ... */);

  static constexpr auto operator()() noexcept {
    return *this;
  }

};

inline constexpr repeat_t repeat{};
```

LSEG


start
Child
Complete


Connect
Arguments
Completion
Sender
Receiver
Context
Start
Operation State

C++ now

2025

Extending std::execution

Implementing Custom Algorithms with Senders & Receivers

Robert Leahy

Operation State

Base classes and member type aliases.

```
template<typename Sender, typename Receiver>
class operation_state :
  inlinable_operation_state<
    operation_state<Sender, Receiver>,
    Receiver>,
  public manual_child_operation_state<
    operation_state<Sender, Receiver>,
    std::env_of_t<Receiver>,
    Sender&>
{
  using receiver_base_ = inlinable_operation_state<
    operation_state,
    Receiver>;
  using receiver_base_::get_receiver;
  using env_ = std::env_of_t<Receiver>;
  using operation_state_base_ = manual_child_operation_state<
    operation_state,
    env_,
    Sender&>;
  Sender sender_;
  bool engaged_{false};
  // ...
};
```

LSEG

Operation State

What're these?

```
template<typename Sender, typename Receiver>
class operation_state :
  inlinable_operation_state<
    operation_state<Sender, Receiver>,
    Receiver>,
  public manual_child_operation_state<
    operation_state<Sender, Receiver>,
    std::env_of_t<Receiver>,
    Sender&>
{
  using receiver_base_ = inlinable_operation_state<
    operation_state,
    Receiver>;
  using receiver_base_::get_receiver;
  using env_ = std::env_of_t<Receiver>;
  using operation_state_base_ = manual_child_operation_state<
    operation_state,
    env_,
    Sender&>;
  Sender sender_;
  bool engaged_{false};
  // ...
};
```

LSEG

D3425R1: Reducing operation-state sizes for subobject child operations

Table of Contents

Operation State

Opt in concept and constructor.

```
template<typename Sender, typename Receiver>
class operation_state : /* ... */ {
  // ...
public:
  using operation_state_concept =
    std::execution::operation_state_t;
  explicit constexpr operation_state(Sender s, Receiver r)
    : receiver_base_(std::move(r)),
      sender_(std::move(s))
  {}
  /* ...
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   */
};
```

LSEG

Operation State

Creates and starts the child operation state.

```
template<typename Sender, typename Receiver>
class operation_state : /* ... */ {
  // ...
  constexpr void start() & noexcept {
    operation_state_base_::construct(sender_);
    engaged_ = true;
    operation_state_base_::start();
  }
  /* ...
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   */
};
```

LSEG

Operation State

Connecting a sender and receiver (which this does internally) can throw exceptions, and the containing function is marked `noexcept` (the `operation_state` concept requires this).

```
template<typename Sender, typename Receiver>
class operation_state : /* ... */ {
  // ...
  constexpr void start() & noexcept {
    operation_state_base_::construct(sender_);
    engaged_ = true;
    operation_state_base_::start();
  }
  /* ...
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   */
};
```

LSEG

Operation State

Catch exceptions and deliver them to the receiver.

```
template<typename Sender, typename Receiver>
class operation_state : /* ... */ {
  // ...
  constexpr void start() & noexcept {
    try {
      operation_state_base_::construct(sender_);
    } catch (...) {
      std::execution::set_error(
        std::move(get_receiver()),
        std::current_exception());
      return;
    }
    engaged_ = true;
    operation_state_base_::start();
  }
  /* ...
   *
   *
   *
   *
   *
   *
   */
};
```

Operation State

If we know statically that connect never throws this is still compiled which means the receiver needs to admit asynchronous exceptions despite the fact we know it'll never be called.

```
template<typename Sender, typename Receiver>
class operation_state : /* ... */ {
  // ...
  constexpr void start() & noexcept {
    try {
      operation_state_base_::construct(sender_);
    } catch (...) {
      std::execution::set_error(
        std::move(get_receiver()),
        std::current_exception());
      return;
    }
    engaged_ = true;
    operation_state_base_::start();
  }
  /* ...
   *
   *
   *
   *
   *
   *
   */
};
```

LSEG

Operation State

Creates and starts the child operation state.

The `if constexpr` is necessary due to the fact when connect doesn't throw sending `set_error` may not compile.

```
template<typename Sender, typename Receiver>
class operation_state : /* ... */ {
  // ...
  constexpr void start() & noexcept {
    if constexpr (is_nothrow_connectable_v<Sender&, env_>) {
      operation_state_base_::construct(sender_);
    } else {
      try {
        operation_state_base_::construct(sender_);
      } catch (...) {
        std::execution::set_error(
          std::move(get_receiver()),
          std::current_exception());
        return;
      }
    }
    engaged_ = true;
    operation_state_base_::start();
  }
  /* ...
   *
   */
};
```

LSEG

Operation State

What's this?

```
template<typename Sender, typename Receiver>
class operation_state : /* ... */ {
  // ...
  constexpr void start() & noexcept {
    if constexpr (is_nothrow_connectable_v<Sender&, env_>) {
      operation_state_base_::construct(sender_);
    } else {
      try {
        operation_state_base_::construct(sender_);
      } catch (...) {
        std::execution::set_error(
          std::move(get_receiver()),
          std::current_exception());
        return;
      }
    }
    engaged_ = true;
    operation_state_base_::start();
  }
  /* ...
   *
   */
};
```

LSEG

When Do You Know `connect` Doesn't Throw?

Document Number: P3388R2
Date: 2025-04-01
Reply-to: Robert Leahy <rleahy@rleahy.ca>
Audience: LWG

Abstract

This paper proposes a change which will enable earlier determination that connecting a certain sender with a certain receiver will never throw an exception.

Background

If `noexcept(execution::connect(sndr, rcvr))` is `true`, then if there exists an expression `rcvr2` such that `is_same_v<decltype(get_env(rcvr2)), decltype(get_env(rcvr))>` is `true` and `noexcept(execution::connect(sndr, rcvr2))` is `false`, the program is ill-formed, no diagnostic required.

[*Note:* This allows determination of whether `connect` throws with only the context of the environment, such as within `get_completion_signatures`. —*end note*]

Does connect Throw?

P3388 enables checking whether **connect** throws using an archetype receiver.

```
namespace detail::is_nothrow_connectable {

template<typename Env>
struct receiver {
  using receiver_concept = std::execution::receiver_t;
  template<typename... Args>
  void set_value(Args&&...) && noexcept;
  template<typename T>
  void set_error(T&&) && noexcept;
  void set_stopped() && noexcept;
  Env get_env() const noexcept;
};

}

template<typename Sender, typename Env>
inline constexpr bool is_nothrow_connectable_v = noexcept(
  std::execution::connect(
    std::declval<Sender>(),
    std::declval<
      detail::is_nothrow_connectable::receiver<Env>>()));
```

LSEG

Operation State

Destroys child operation state if it's within its lifetime.

```
template<typename Sender, typename Receiver>
class operation_state : /* ... */ {
  // ...
  constexpr ~operation_state() noexcept {
    if (engaged_) {
      operation_state_base_::destruct();
    }
  }
  /* ...
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   */
};
```

LSEG

Operation State

When the child operation completes successfully we loop.

```
template<typename Sender, typename Receiver>
class operation_state : /* ... */ {
  // ...
  constexpr void set_value() noexcept {
    operation_state_base_::destruct();
    engaged_ = false;
    start();
  }
  /* ...
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   */
};
```

LSEG

Operation State

Error and stopped lead to the overall operation ending with that error or with stopped.

```
template<typename Sender, typename Receiver>
class operation_state : /* ... */ {
  // ...
  template<typename... Args>
  constexpr void set_error(Args&&... args) noexcept {
    std::execution::set_error(
      std::move(get_receiver()),
      std::forward<Args>(args)...);
  }
  template<typename... Args>
  constexpr void set_stopped(Args&&... args) noexcept {
    std::execution::set_stopped(
      std::move(get_receiver()),
      std::forward<Args>(args)...);
  }
  /* ...
   *
   *
   *
   *
   *
   *
   */
};
```

LSEG

Operation State

Passes the environment through from our receiver to our child.

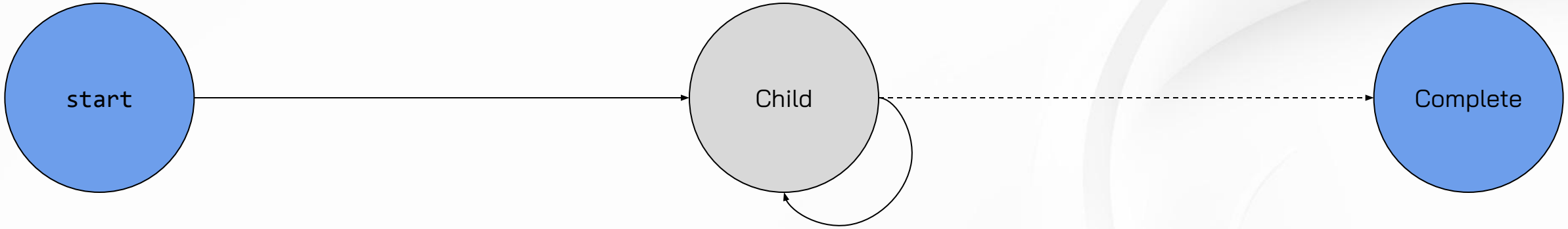
```
template<typename Sender, typename Receiver>
class operation_state : /* ... */ {
  // ...
  constexpr env_ get_env() noexcept {
    return std::get_env(receiver_base_::get_receiver());
  }
};
```

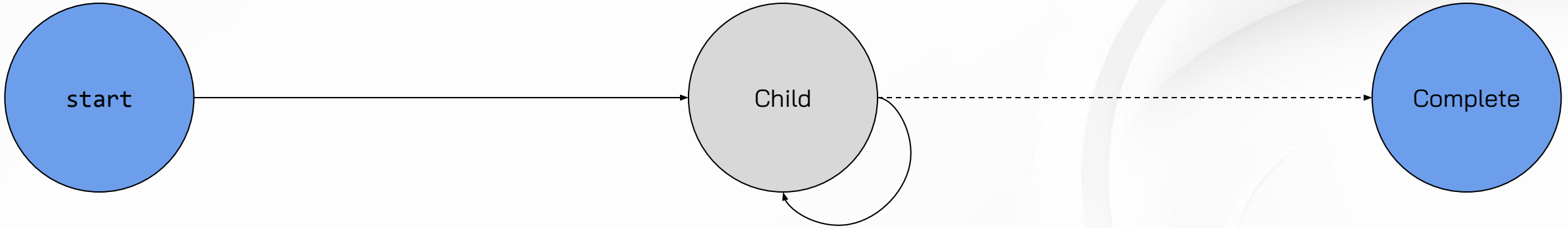
Asynchronous Network Server

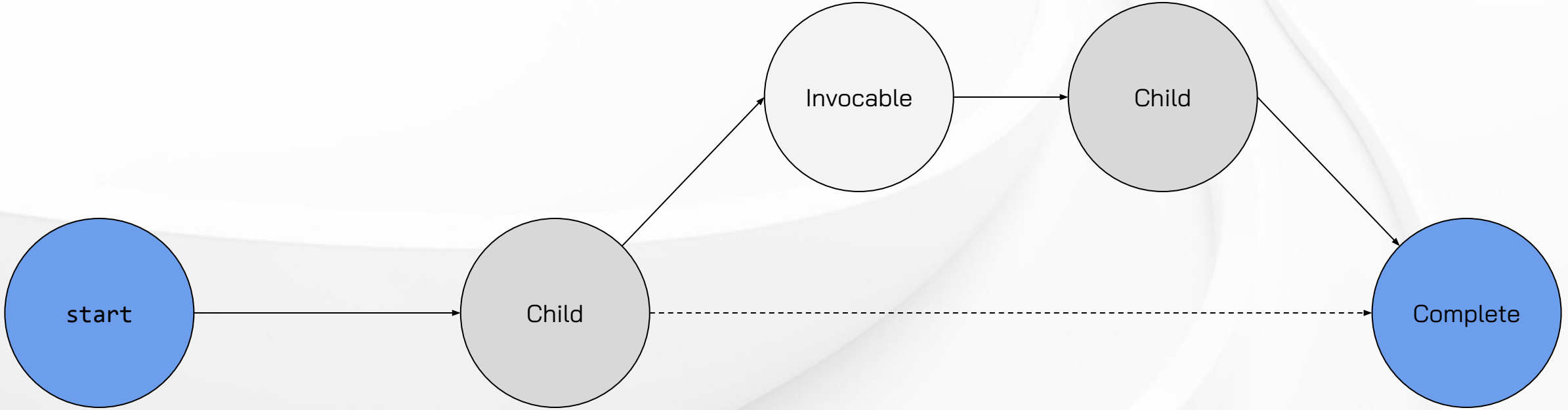
Are we done?

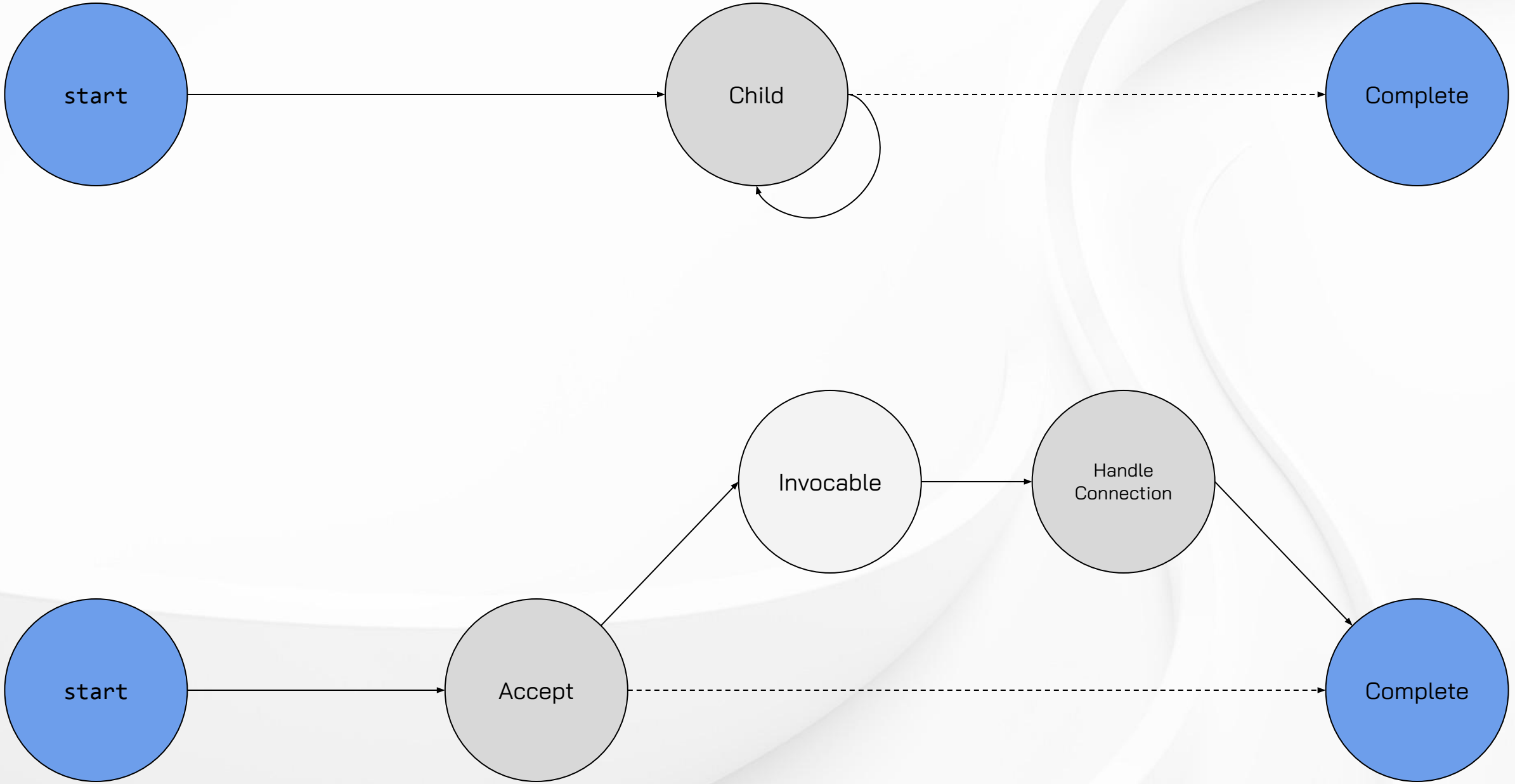
```
std::execution::sender auto accept(int fd);

std::execution::sender auto handle_connection(int fd);

const int accepting = /* ... */;
std::execution::sender auto server =
  accept(accepting) |
  std::execution::let_value(handle_connection) |
  repeat();
```

LSEG

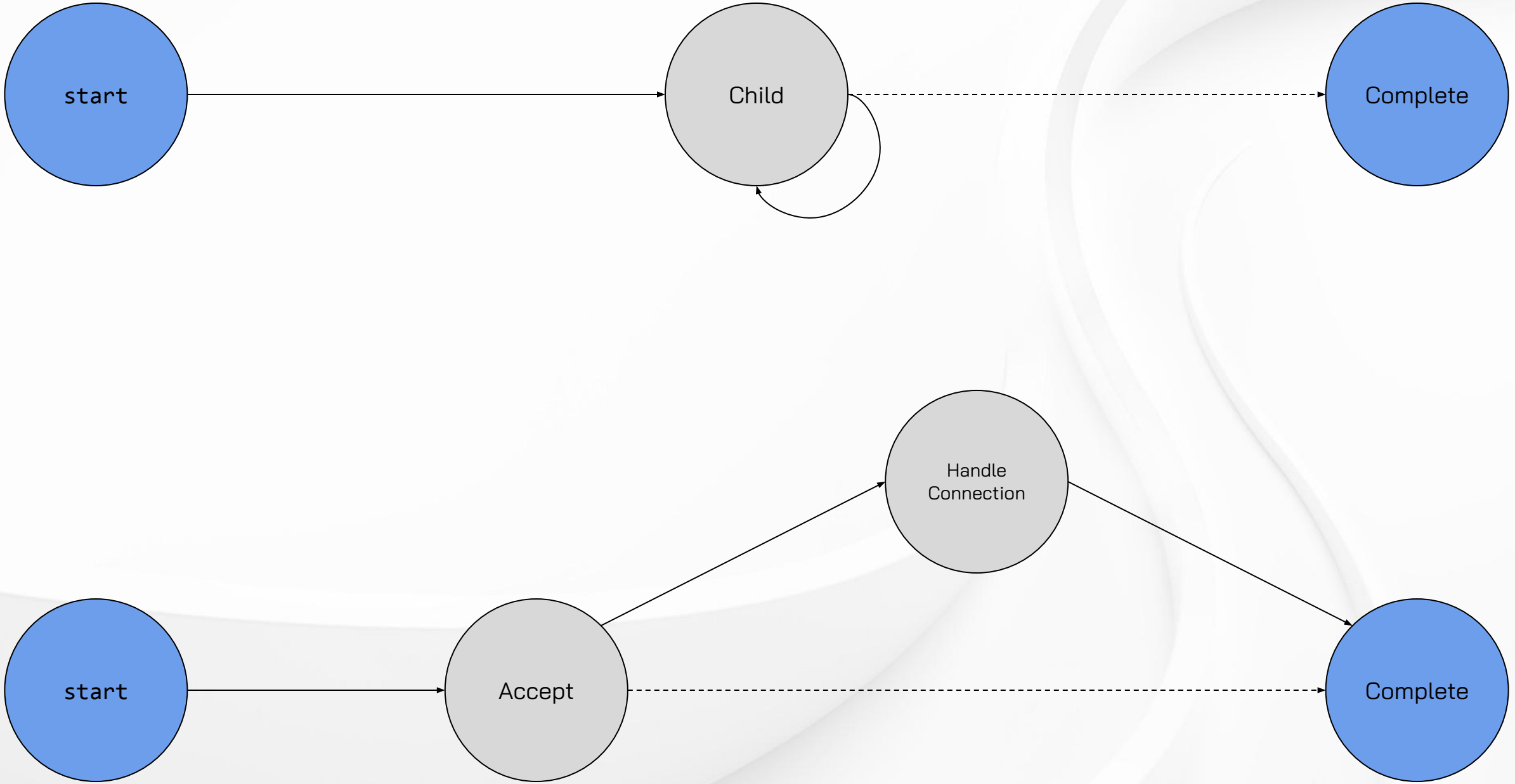

start
Child
Complete

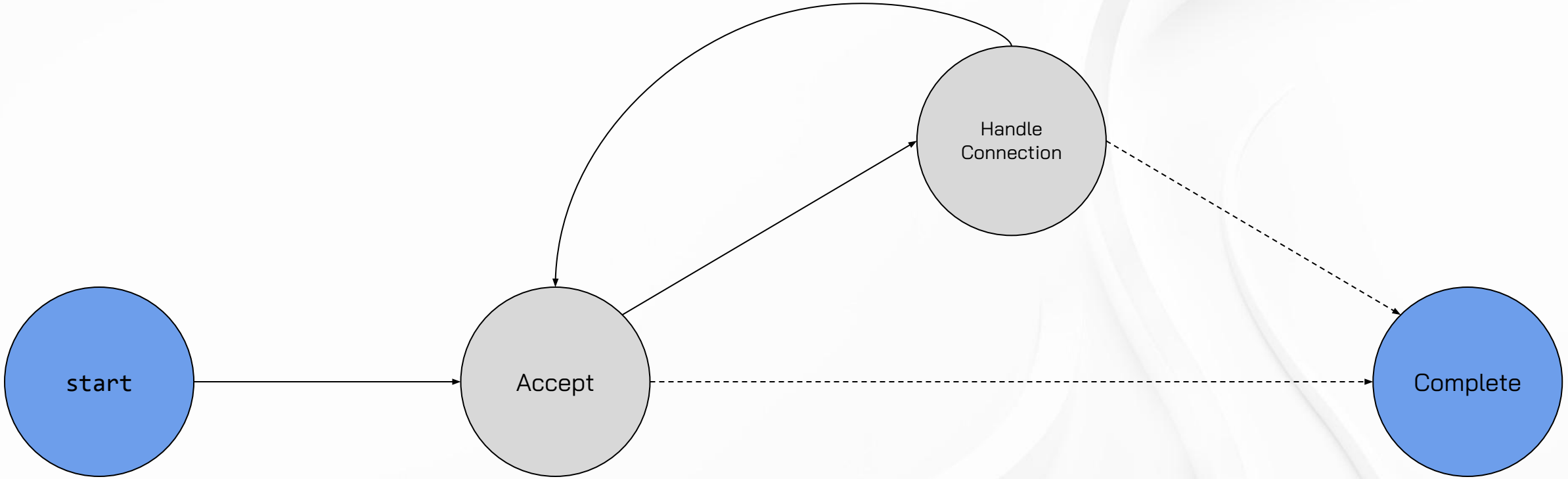

start
Child
Complete

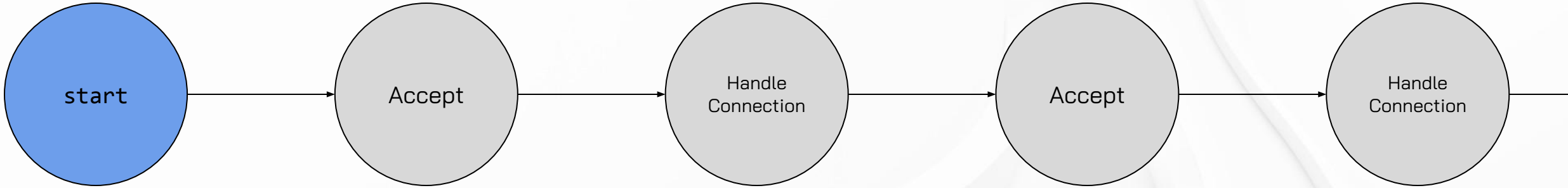

Invocable
Child
start
Child
Complete


start
Child
Complete
Invocable
Handle
Connection
start
Accept
Complete

LSEG


start
Child
Complete
Handle
Connection
start
Accept
Complete


start
Accept
Handle
Connection
Complete


start
Accept
Handle Connection
Accept
Handle Connection

Network Server

Synchronous code from earlier in the talk which has the same issue assuming `handle_connection` is blocking.

```
const int accepting = /* ... */;
for (;;) {
  const auto conn = accept(accepting, nullptr, nullptr);
  if (conn == -1) {
    throw std::system_error(/* ... */);
  }
  handle_connection(conn);
}
```

Network Server

Handles connections in the background.

```
const int accepting = /* ... */;
for (;;) {
  const auto conn = accept(accepting, nullptr, nullptr);
  if (conn == -1) {
    throw std::system_error(/* ... */);
  }
  std::thread(handle_connection, conn).detach();
}
```

Network Server

Who waits for all these detached threads?

```
const int accepting = /* ... */;
for (;;) {
  const auto conn = accept(accepting, nullptr, nullptr);
  if (conn == -1) {
    throw std::system_error(/* ... */);
  }
  std::thread(handle_connection, conn).detach();
}
```

LSEG

Network Server

Updated to track and wait for detached threads.

```
std::mutex m;
std::condition_variable cv;
std::size_t connections = 0;
const auto wait = [&]() noexcept {
  std::unique_lock l(m);
  cv.wait(l, [&]() noexcept { return !connections; });
};
const int accepting = /* ... */;
for (;;) {
  const auto conn = accept(accepting, nullptr, nullptr);
  if (conn == -1) {
    throw std::system_error(/* ... */);
  }
  const std::lock_guard l(m);
  ++connections;
  std::thread([&, conn]() {
    handle_connection(conn);
    const std::lock_guard l(m);
    if (!--connections) cv.notify_one();
  }).detach();
}
wait();
```

Network Server

What if these throw?

```
std::mutex m;
std::condition_variable cv;
std::size_t connections = 0;
const auto wait = [&]() noexcept {
  std::unique_lock l(m);
  cv.wait(l, [&]() noexcept { return !connections; });
};
const int accepting = /* ... */;
for (;;) {
  const auto conn = accept(accepting, nullptr, nullptr);
  if (conn == -1) {
    throw std::system_error(/* ... */);
  }
  const std::lock_guard l(m);
  ++connections;
  std::thread([&, conn]() {
    handle_connection(conn);
    const std::lock_guard l(m);
    if (!--connections) cv.notify_one();
  }).detach();
}
wait();
```

LSEG

Network Server

Waits and then rethrows.

Note that fitting all this on the slide required that the font size be reduced.

```
std::mutex m;
std::condition_variable cv;
std::size_t connections = 0;
const auto wait = [&]() noexcept {
  std::unique_lock l(m);
  cv.wait(l, [&]() noexcept { return !connections; });
};
const int accepting = /* ... */;
for (;;) {
  try {
    const auto conn = accept(accepting, nullptr, nullptr);
    if (conn == -1) {
      throw std::system_error(/* ... */);
    }
    const std::lock_guard l(m);
    ++connections;
    std::thread([&, conn]() {
      handle_connection(conn);
      const std::lock_guard l(m);
      if (!--connections) cv.notify_one();
    }).detach();
  } catch (...) {
    wait();
    throw;
  }
}
wait();
```

LSEG

Network Server

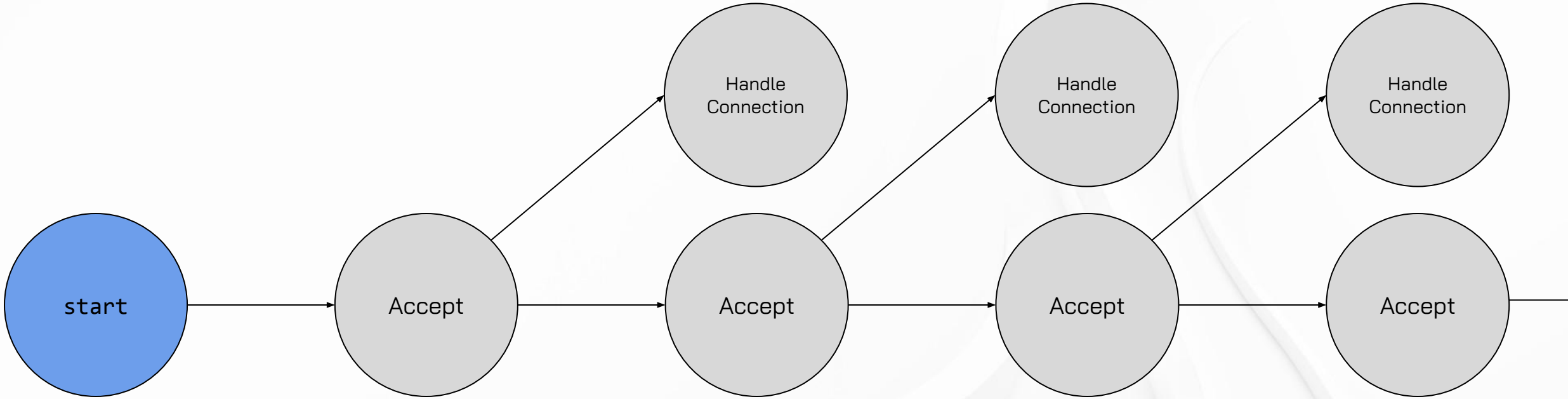
What if this throws?

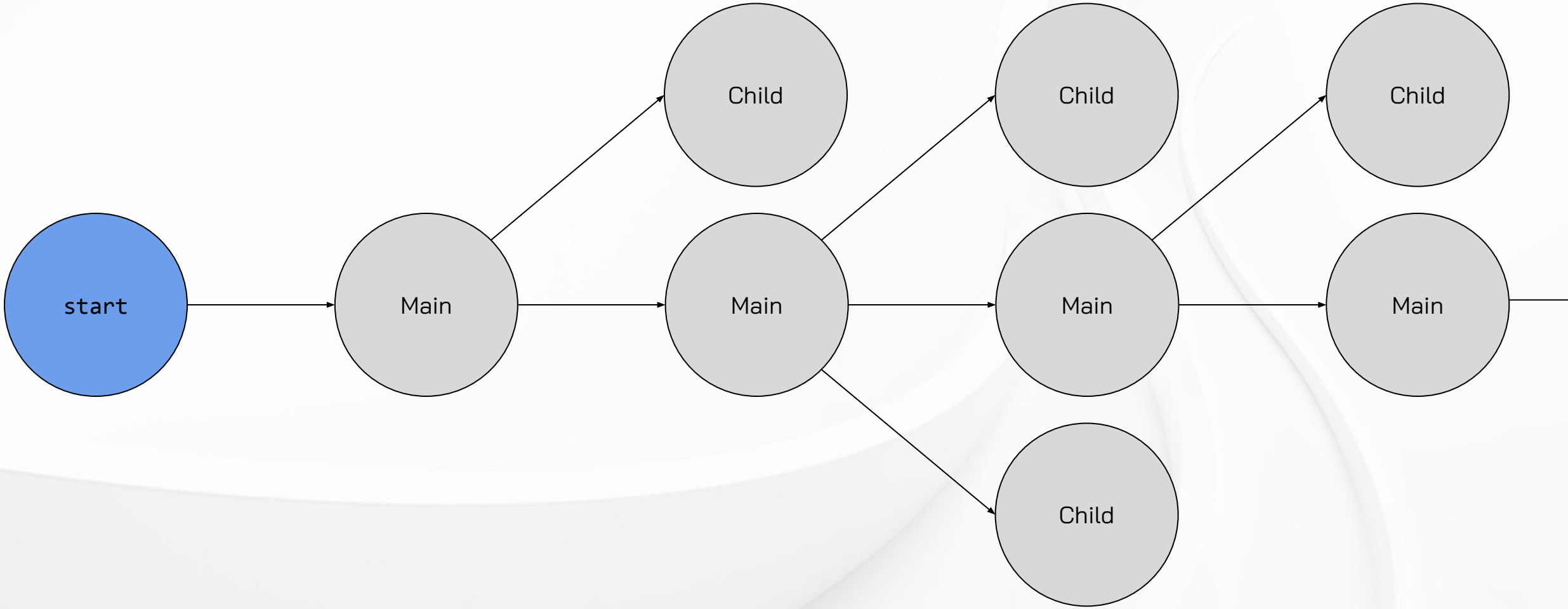
```
std::mutex m;
std::condition_variable cv;
std::size_t connections = 0;
const auto wait = [&]() noexcept {
  std::unique_lock l(m);
  cv.wait(l, [&]() noexcept { return !connections; });
};
const int accepting = /* ... */;
for (;;) {
  try {
    const auto conn = accept(accepting, nullptr, nullptr);
    if (conn == -1) {
      throw std::system_error(/* ... */);
    }
    const std::lock_guard l(m);
    ++connections;
    std::thread([&, conn]() {
      handle_connection(conn);
      const std::lock_guard l(m);
      if (!--connections) cv.notify_one();
    }).detach();
  } catch (...) {
    wait();
    throw;
  }
}
wait();
```

LSEG

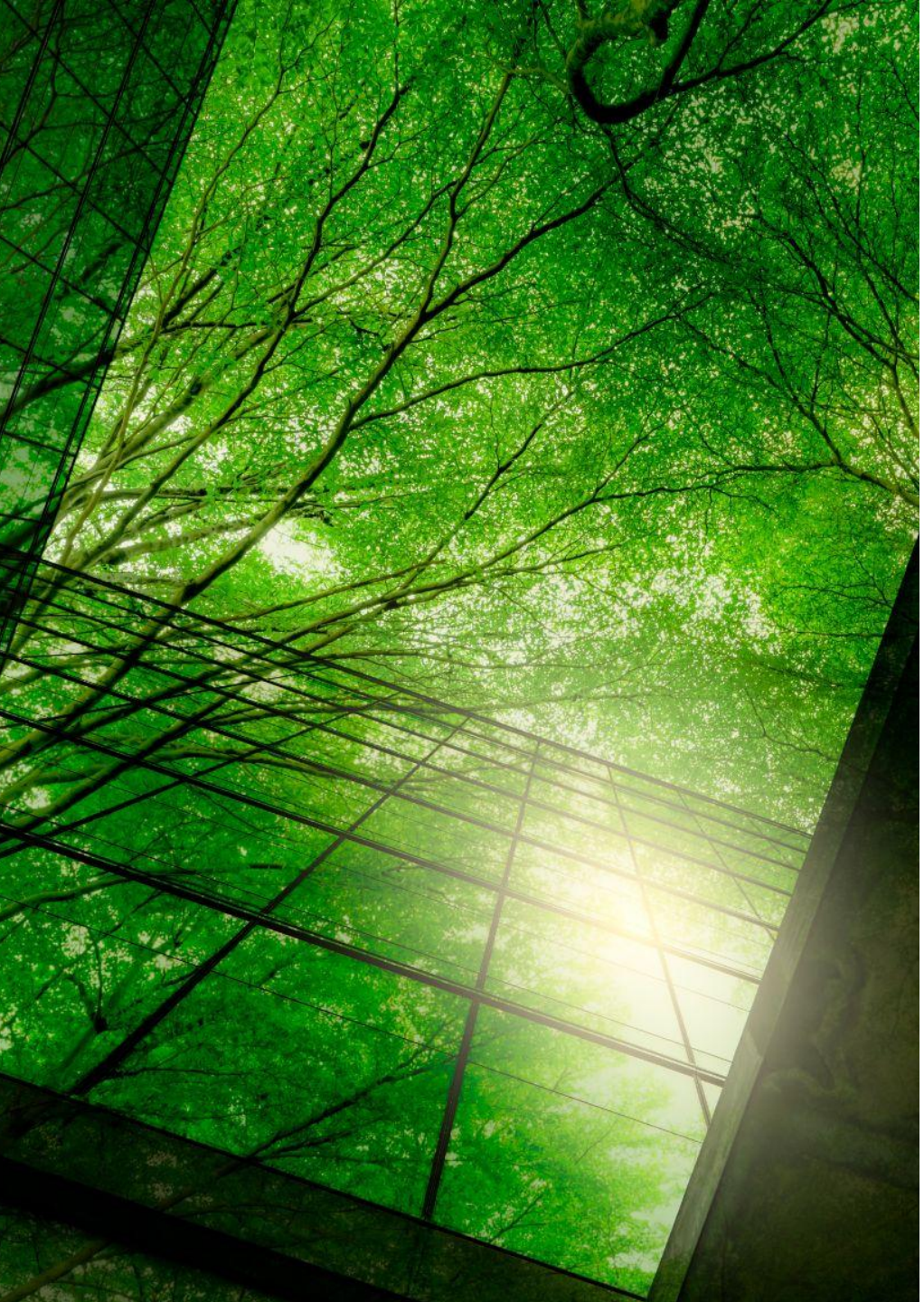
Is this good code?

LSEG


start
Accept
Handle Connection
Accept
Handle Connection
Accept
Handle Connection
Accept


start
Main
Child
Main
Child
Main
Child
Main
Child
Main

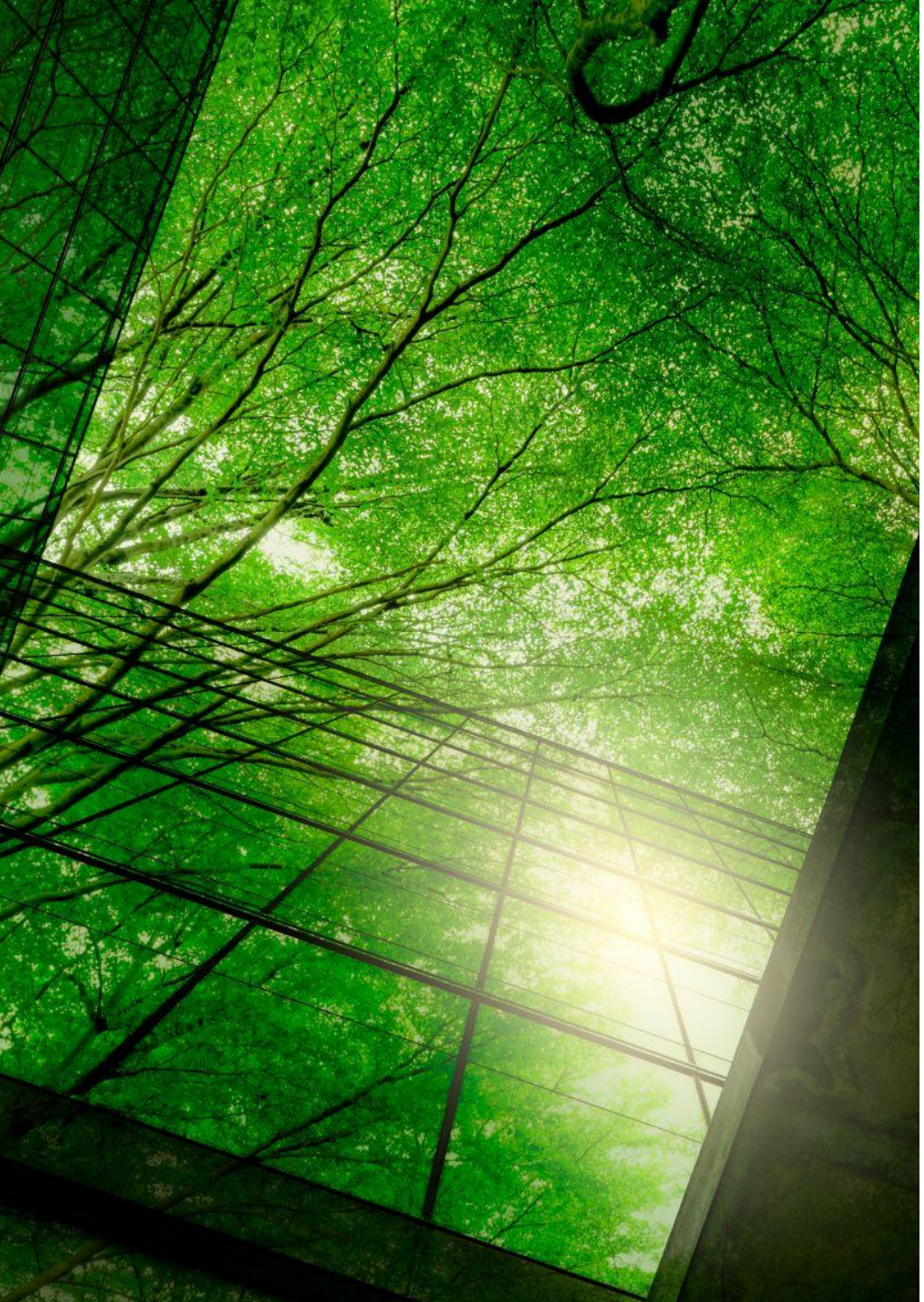

branch



branch

Dynamic fan out

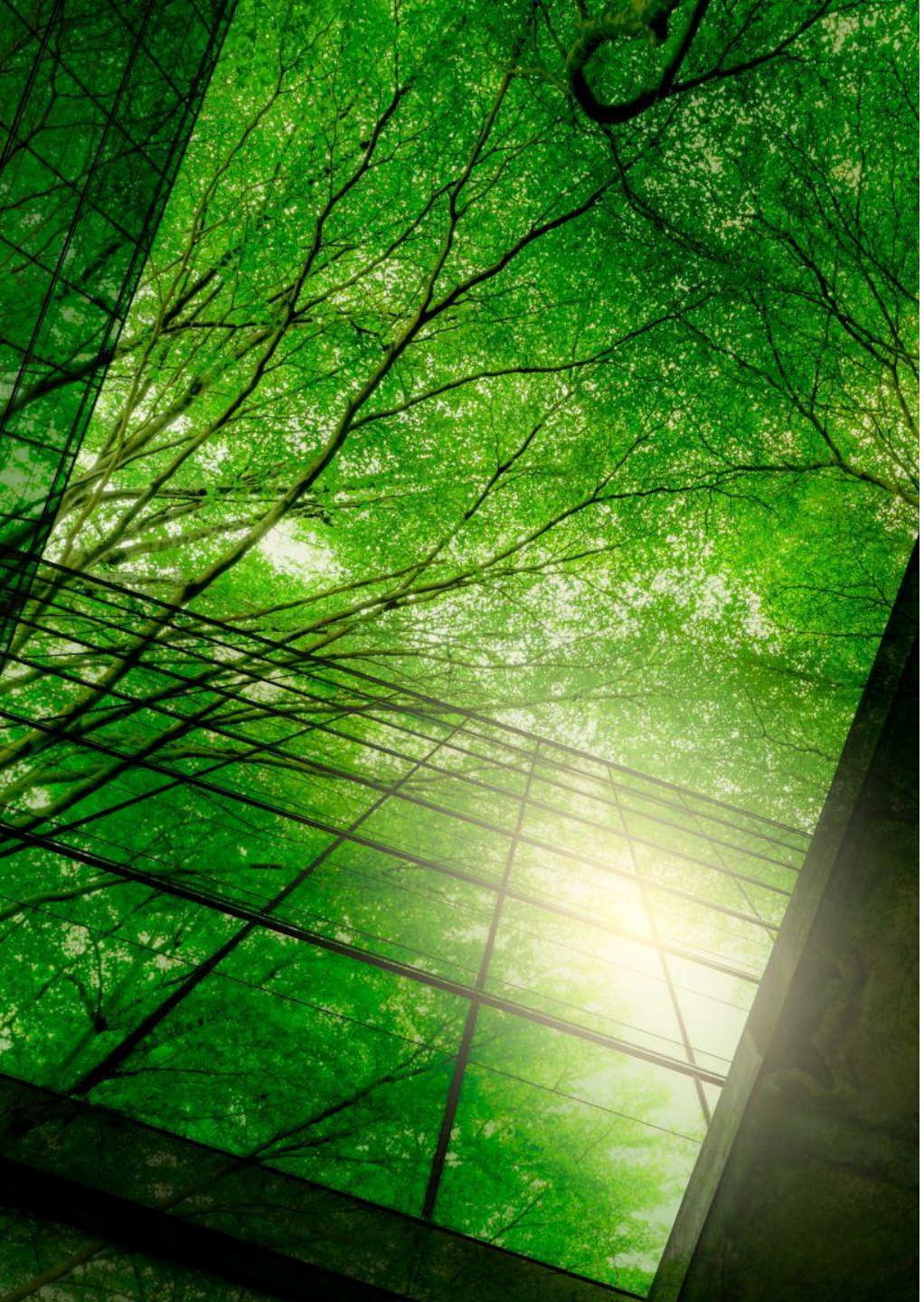
- Accepts a single, multishot sender (the “main sender”) whose completion signatures may include
  - `std::execution::set_value_t()`
  - `std::execution::set_error_t(T)`
  - `std::execution::set_stopped_t()`
  - `std::execution::set_value_t(Senders...)`



`branch`

Dynamic fan out

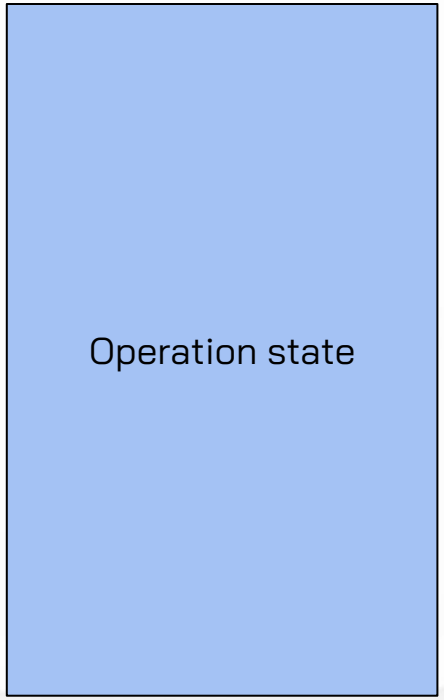
- Accepts a single, multishot sender (the “main sender”) whose completion signatures may include
  - `std::execution::set_value_t()`
  - `std::execution::set_error_t(T)`
  - `std::execution::set_stopped_t()`
  - `std::execution::set_value_t(Senders...)`
- When the returned sender (the “`branch` sender”) is connected and started
  - Connects and starts the main sender
  - Upon the completion thereof begins fan in if the received signature is not `std::execution::set_value_t(Senders...)`, otherwise
  - Connects and starts each received sender (henceforth referred to as “child senders”)
  - Restarts from the first step

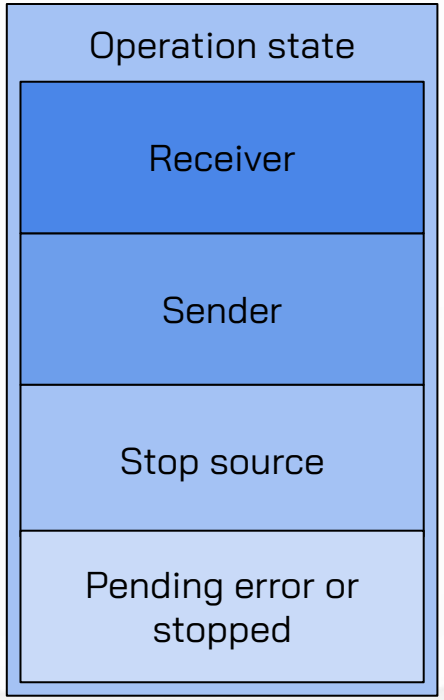


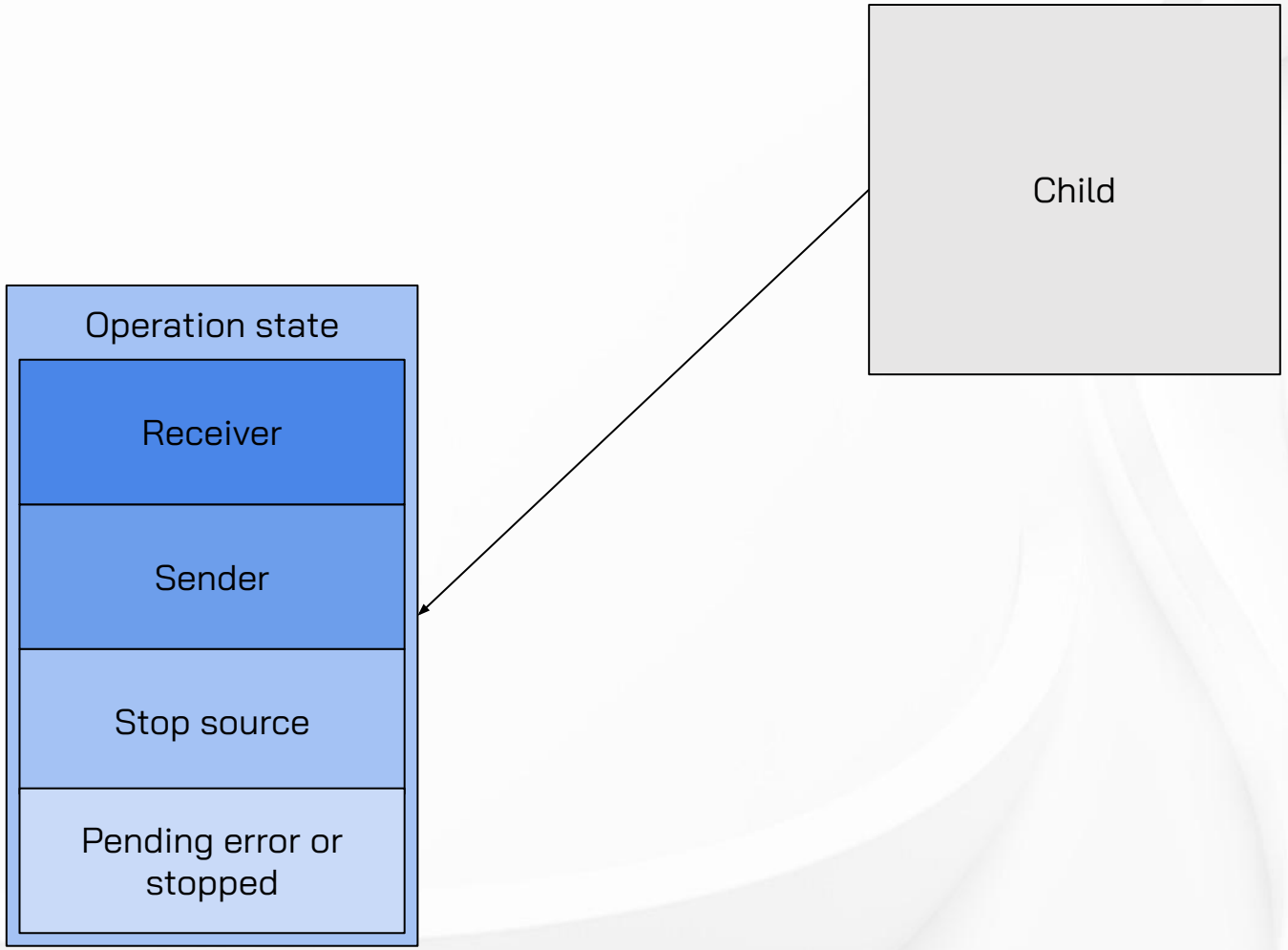
`branch`

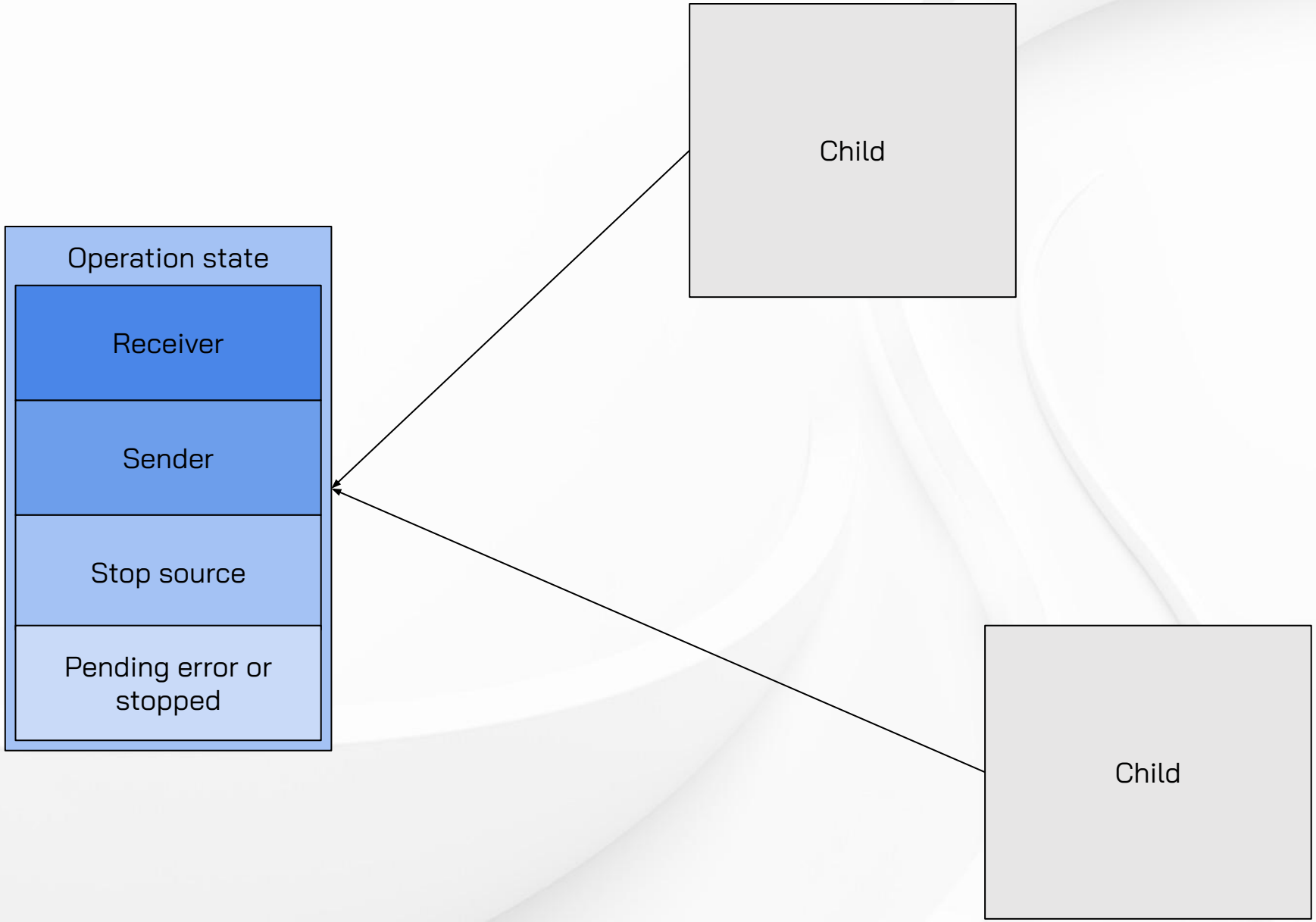
Dynamic fan out

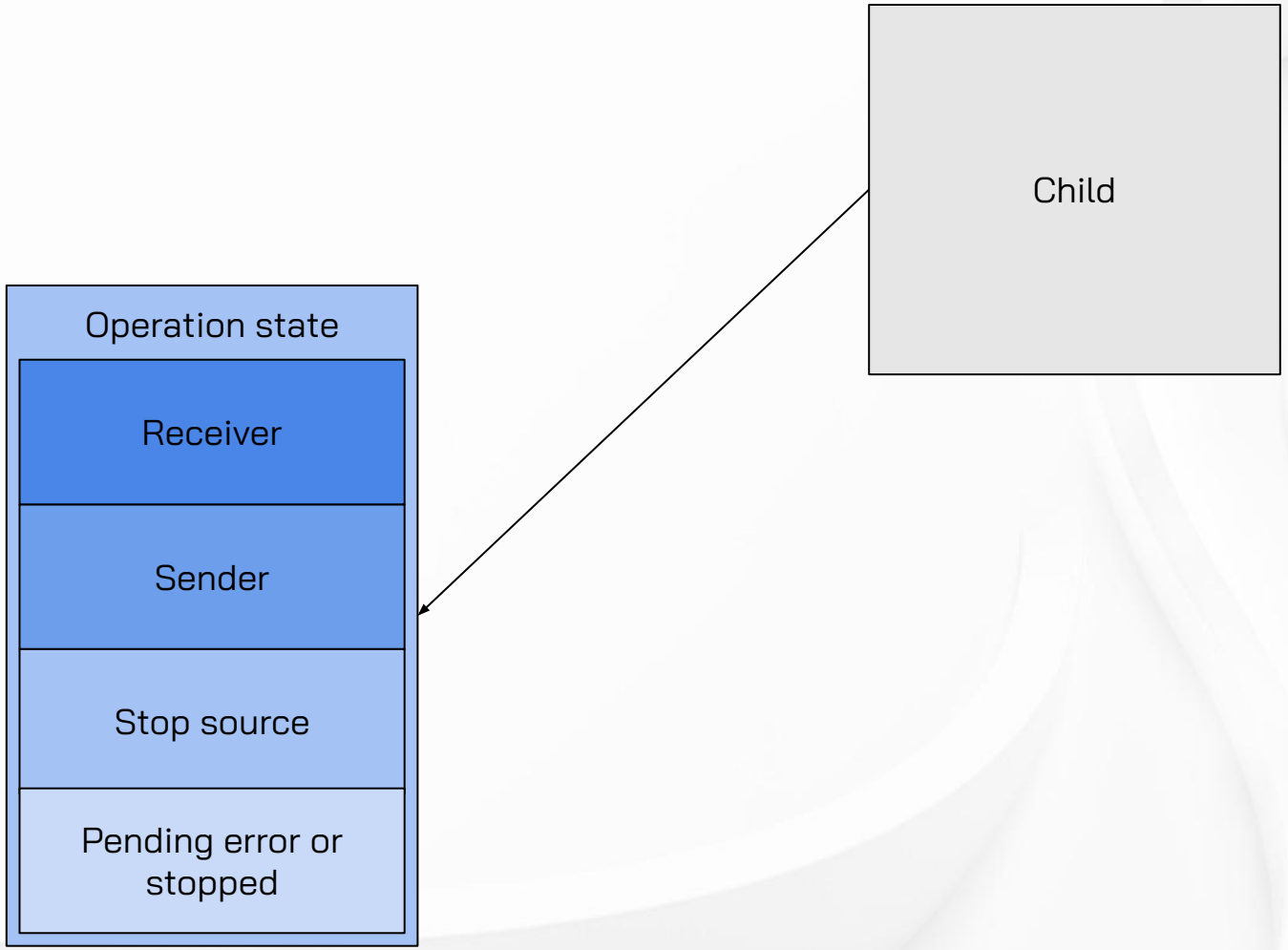
- Accepts a single, multishot sender (the “main sender”) whose completion signatures may include
  - `std::execution::set_value_t()`
  - `std::execution::set_error_t(T)`
  - `std::execution::set_stopped_t()`
  - `std::execution::set_value_t(Senders...)`
- When the returned sender (the “`branch` sender”) is connected and started
  - Connects and starts the main sender
  - Upon the completion thereof begins fan in if the received signature is not `std::execution::set_value_t(Senders...)`, otherwise
  - Connects and starts each received sender (henceforth referred to as “child senders”)
  - Restarts from the first step
- When a child sender completes
  - Begins fan in if the received signature is
    - `std::execution::set_error_t(T)`
    - `std::execution::set_stopped_t()`
  - Otherwise all other execution simply continues

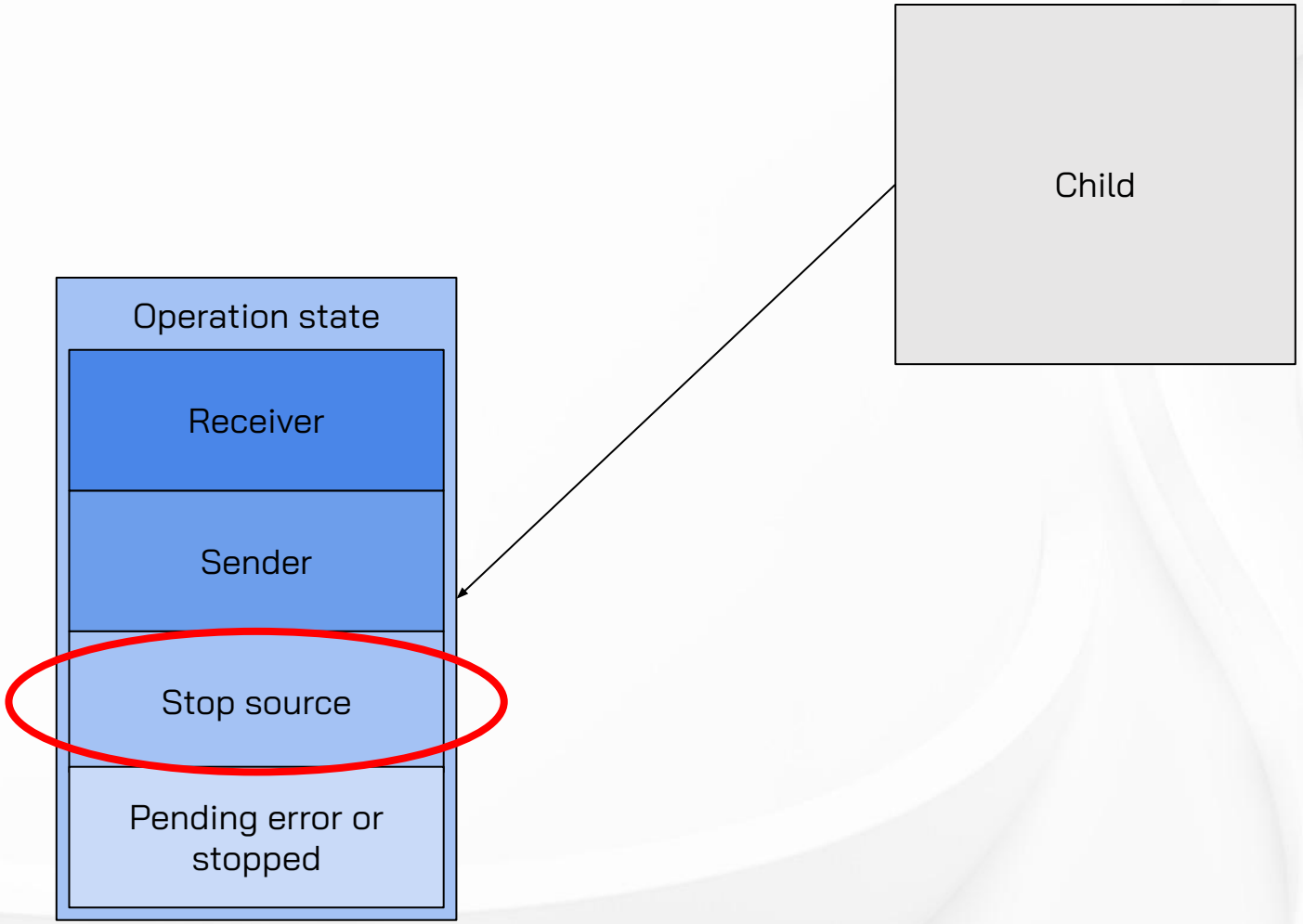

Operation state


Operation state
Receiver
Sender
Stop source
Pending error or stopped


Child
Operation state
Receiver
Sender
Stop source
Pending error or stopped


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


Child
Operation state
Receiver
Sender
Stop source
Pending error or stopped


Child
Operation state
Receiver
Sender
Stop source
Pending error or stopped

chained_stop_source

As presented in C++ Now talk.

Forwards stop requests through `token` into the `inplace_stop_source` from which it derives.

```
template<std::stoppable_token Token>
struct chained_stop_source : std::inplace_stop_source {

  constexpr explicit chained_stop_source(const Token& token);

};
```

Let `rcvr` be a receiver and let `op_state` be an operation state associated with an asynchronous operation created by connecting `rcvr` with a sender. Let `token` be a stop token equal to `get_stop_token(get_env(rcvr))`. `token` shall remain valid for the duration of the asynchronous operation's lifetime ([exec.async.ops]).

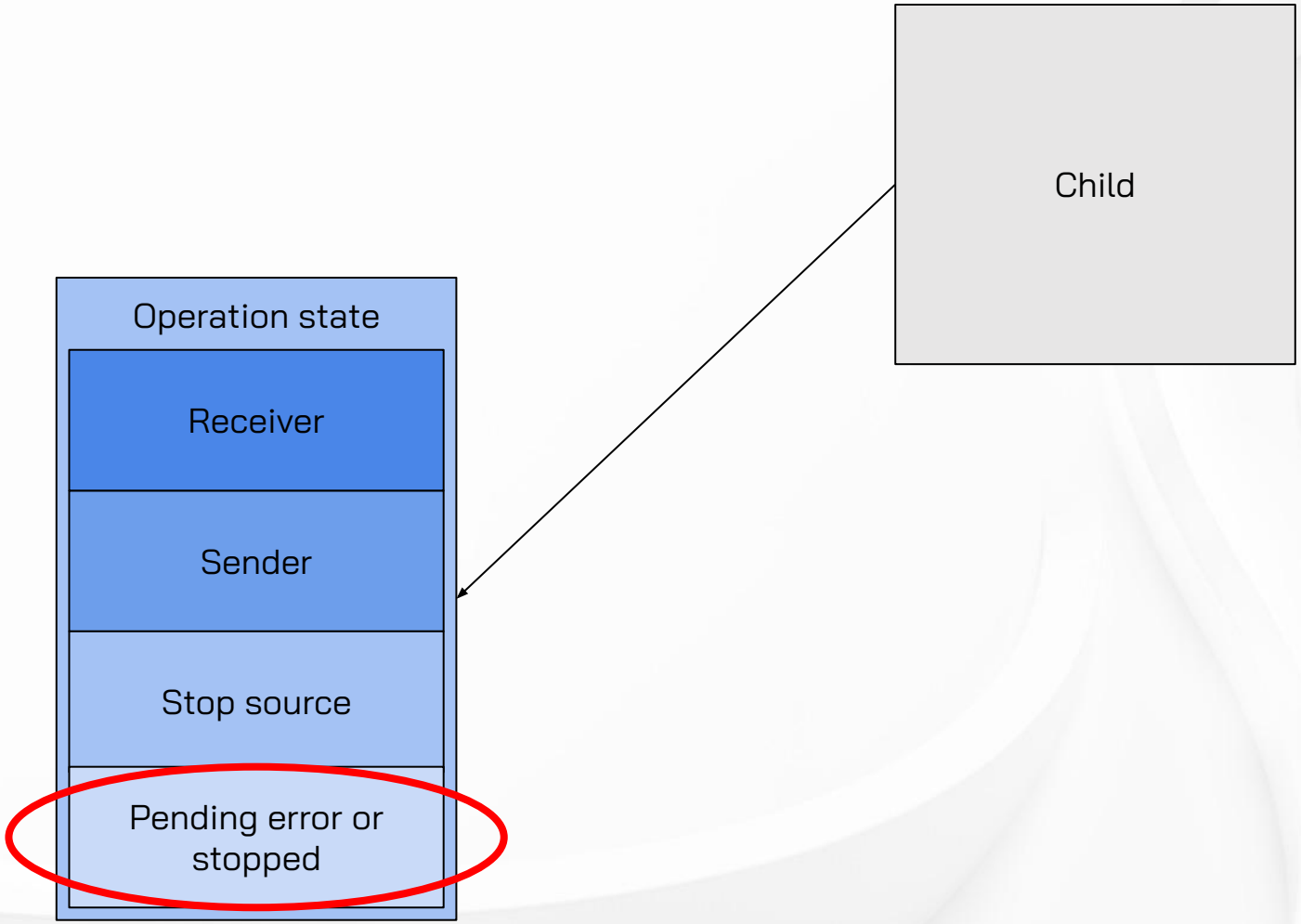
[*Note 1*: This means that, unless it knows about further guarantees provided by the type of `rcvr`, the implementation of `op_state` cannot use `token` after it executes a completion operation. This also implies that any stop callbacks registered on token must be destroyed before the invocation of the completion operation. — *end note*]

chained_stop_source

Updated API. attach starts using `token`, `detach` stops using it.

This sort of manual management of lifetimes is common in `std::execution`-based code due to the confluence of synchronous and asynchronous lifetimes.

```
template<std::stoppable_token Token>
struct chained_stop_source : std::inplace_stop_source {

  constexpr explicit chained_stop_source() noexcept;

  constexpr void attach(const Token& token);

  constexpr void detach() noexcept;

};
```


Child
Operation state
Receiver
Sender
Stop source
Pending error or stopped

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`.

```
template<typename Signatures>
struct storage_for_completion_signatures {

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

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

};
```

LSEG

Completion Storage

What is this?

```
template<typename Signatures>
struct storage_for_completion_signatures {

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

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

};
```

LSEG

Asynchronous Function Signature

Communicates the ways in which the asynchronous operation created from a certain sender can complete.

```
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()>;
```

LSEG

Completion Signatures

Let's suppose the main sender provided to **branch** has this set of completion signatures.

```
std::execution::completion_signatures<
  std::execution::set_value_t(),
  std::execution::set_value_t(my_sender)>;
```

Completion Signatures

Since the main sender can send nullary `set_value` the overall operation can thusly complete.

Also since we'll need to perform allocations to spawn children we'll need to be able to asynchronously throw exceptions in case of allocation failure.

```
std::execution::completion_signatures<
  std::execution::set_value_t(),
  std::execution::set_value_t(my_sender)>;
```



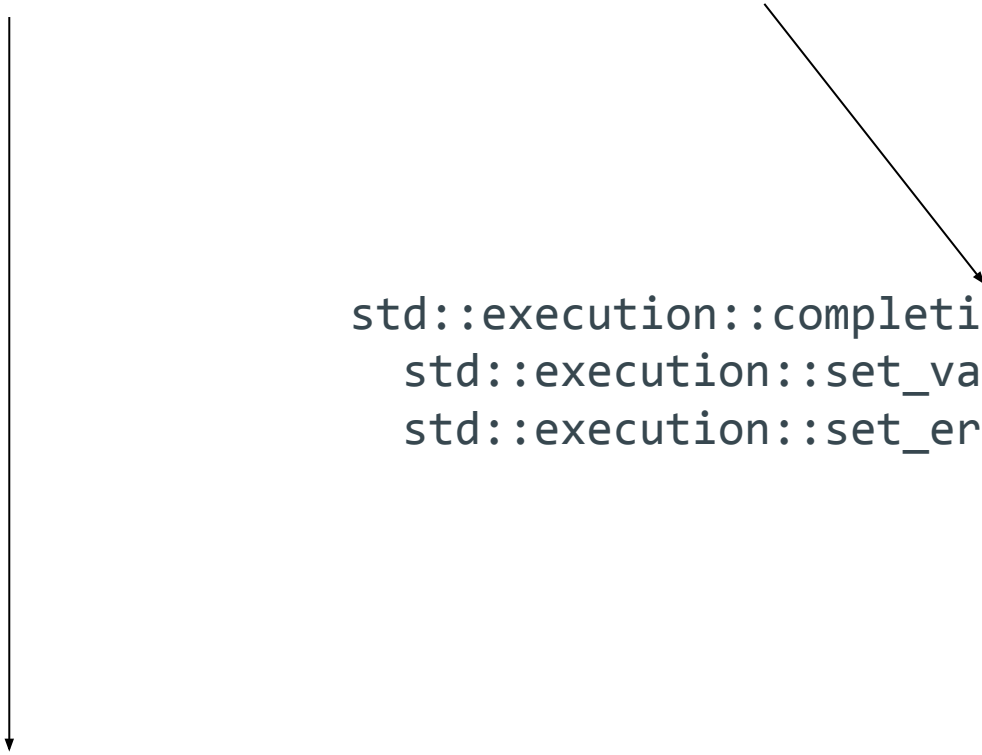
```
std::execution::completion_signatures<
  std::execution::set_value_t(),
  std::execution::set_error_t(std::exception_ptr)>;
```

LSEG

Completion Signatures

Now let's imagine `my_sender` has this set of completion signatures.

```
std::execution::completion_signatures<
  std::execution::set_value_t(),
  std::execution::set_value_t(my_sender)>;
```



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

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

LSEG

Completion Signatures

We need to be able to pass through errors from the child, therefore we need to add a new `set_error` signature to our output set of completion signatures.

```
std::execution::completion_signatures<
  std::execution::set_value_t(),
  std::execution::set_value_t(my_sender)>;
```

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

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

LSEG


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.

Previously presented at C++Now 2025.

```
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

Since we're going to be allocating we start with a set of signatures that contains an asynchronous exception throw.

Since completion signatures need to be unique we have a helper to ensure uniqueness.

```
consteval auto completion_signatures(
  const std::meta::info signatures,
  const std::meta::info env)
{
  std::vector<std::meta::info> results{
    ^^std::execution::set_error_t(std::exception_ptr)};
  const auto add = [&](const std::meta::info info) {
    const auto iter = std::ranges::find(results, info);
    if (iter == results.end()) {
      results.push_back(info);
    }
  };
  /* ...
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   */
}
```

LSEG

Completion Signatures

We loop over and process each signature.

If we don't find any senders we fail: We can't branch if there's nothing to branch to.

```
consteval auto completion_signatures(
  const std::meta::info signatures,
  const std::meta::info env)
{
  std::vector<std::meta::info> results{
    ^^std::execution::set_error_t(std::exception_ptr)};
  const auto add = [&](const std::meta::info info) {
    const auto iter = std::ranges::find(results, info);
    if (iter == results.end()) {
      results.push_back(info);
    }
  };
  bool found_sender = false;
  // ...
  const auto signature = [&](std::meta::info) { /* ... */ };
  for (auto&& s : template_arguments_of(signatures)) {
    signature(s);
  }
  if (!found_sender) {
    throw /* ... */;
  }
  return substitute(
    ^^std::execution::completion_signatures,
    results);
}
```

LSEG

Completion Signatures

What's this?

```
consteval auto completion_signatures(
  const std::meta::info signatures,
  const std::meta::info env)
{
  std::vector<std::meta::info> results{
    ^^std::execution::set_error_t(std::exception_ptr)};
  const auto add = [&](const std::meta::info info) {
    const auto iter = std::ranges::find(results, info);
    if (iter == results.end()) {
      results.push_back(info);
    }
  };
  bool found_sender = false;
  // ...
  const auto signature = [&](std::meta::info) { /* ... */ };
  for (auto&& s : template_arguments_of(signatures)) {
    signature(s);
  }
  if (!found_sender) {
    throw /* ... */;
  }
  return substitute(
    ^^std::execution::completion_signatures,
    results);
}
```

LSEG

Completion Signatures

If the signature is sent on the value channel we inspect the parameters, if there are none this indicates that branch can end, so we add that, otherwise each parameter represents a potential child sender, so we visit each of them.

If the signature is sent on any other channel we simply copy it (with some transformations as we'll see later).

```
consteval auto completion_signatures(
  const std::meta::info signatures,
  const std::meta::info env)
{
  // ...
  const auto copy_signature = [&](std::meta::info) { /* ... */ };
  const auto child = [&](std::meta::info) { /* ... */ };
  const auto signature = [&](const std::meta::info sig) {
    if (return_type_of(sig) == ^^std::execution::set_value_t) {
      const auto params = parameters_of(sig);
      if (params.empty()) {
        add(sig);
        return;
      }
      found_sender = true;
      for (auto&& param : params) {
        child(param);
      }
    } else {
      copy_signature(sig);
    }
  };
  /* ...
   */
}
```

LSEG

Completion Signatures

Time to review more lambdas.

```
consteval auto completion_signatures(
  const std::meta::info signatures,
  const std::meta::info env)
{
  // ...
  const auto copy_signature = [&](std::meta::info) { /* ... */ };
  const auto child = [&](std::meta::info) { /* ... */ };
  const auto signature = [&](const std::meta::info sig) {
    if (return_type_of(sig) == ^^std::execution::set_value_t) {
      const auto params = parameters_of(sig);
      if (params.empty()) {
        add(sig);
        return;
      }
      found_sender = true;
      for (auto&& param : params) {
        child(param);
      }
    } else {
      copy_signature(sig);
    }
  };
  /* ...
   */
}
```

LSEG

Completion Signatures

`set_stopped` never has parameters, so we just add it.

Parameters of `set_error` must be decayed and decayable because we'll be storing them in the operation state and then sending them later once fan in is complete.

```
consteval auto completion_signatures(
  const std::meta::info signatures,
  const std::meta::info env)
{
  // ...
  const auto copy_signature = [&](const std::meta::info sig) {
    if (return_type_of(sig) == ^^std::execution::set_stopped_t) {
      add(sig);
    } else {
      const auto param = parameters_of(sig).at(0);
      const auto decayed = decay(param);
      if (!is_constructible_type(decayed, {param})) {
        throw /* ... */;
      }
      add(make_completion_signature(tag, {decayed}));
    }
  };
  /* ...
   *
   *
   *
   *
   *
   *
   */
}
```

LSEG

Completion Signatures

Another lambda to review.

```
consteval auto completion_signatures(
  const std::meta::info signatures,
  const std::meta::info env)
{
  // ...
  const auto copy_signature = [&](std::meta::info) { /* ... */ };
  const auto child = [&](std::meta::info) { /* ... */ };
  const auto signature = [&](const std::meta::info sig) {
    if (return_type_of(sig) == ^^std::execution::set_value_t) {
      const auto params = parameters_of(sig);
      if (params.empty()) {
        add(sig);
        return;
      }
      found_sender = true;
      for (auto&& param : params) {
        child(param);
      }
    } else {
      copy_signature(sig);
    }
  };
  /* ...
   */
}
```

LSEG

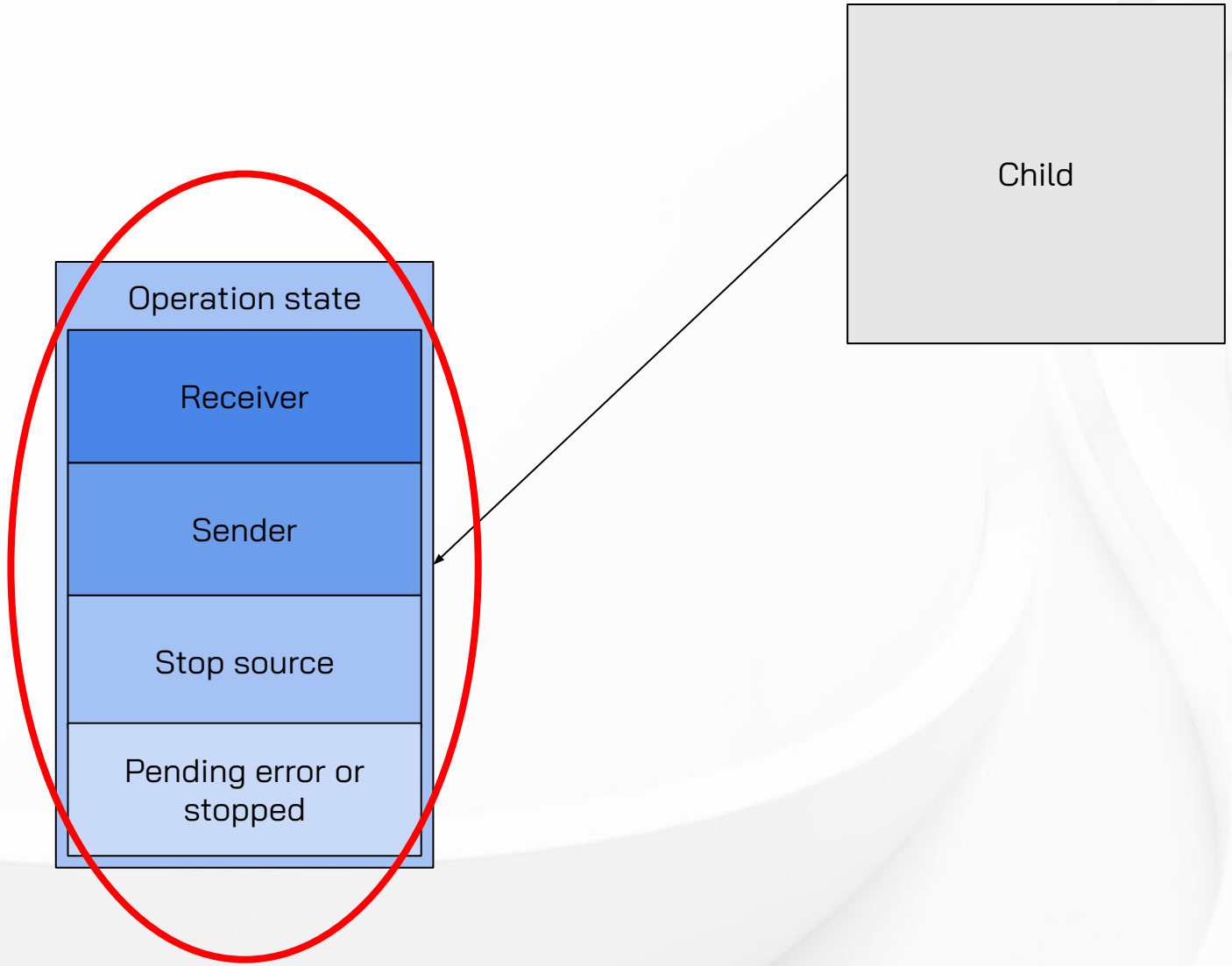
Completion Signatures

Obtain the completion signatures of the child and process each of them.

`set_value` signatures with parameters lead to rejection because we don't have a way to deal with the arguments.

```
consteval auto completion_signatures(
  const std::meta::info signatures,
  const std::meta::info env)
{
  // ...
  const auto child = [&](const std::meta::info sender) {
    const auto child_signatures = dealias(
      substitute(
        ^^std::execution::completion_signatures_of_t,
        {sender, env}));
    for (auto&& sig : template_arguments_of(child_signatures)) {
      if (return_type_of(sig) == ^^std::execution::set_value_t) {
        if (!parameters_of(signature).empty()) {
          throw /* ... */;
        }
      } else {
        copy_signature(sig);
      }
    }
  };
  /* ...
   *
   *
   */
}
```

LSEG


Child
Operation state
Receiver
Sender
Stop source
Pending error or stopped

Operation State

Base classes and member type aliases.

```
template<typename Sender, typename Receiver>
class operation_state :
  inlinable_operation_state<
    operation_state<Sender, Receiver>,
    Receiver>,
  public manual_child_operation_state<
    operation_state<Sender, Receiver>,
    std::env_of_t<Receiver>,
    Sender&>
{
  using receiver_base_ = inlinable_operation_state<
    operation_state,
    Receiver>;
  using receiver_base_::get_receiver;
  using receiver_env_ = std::env_of_t<Receiver>;
  using env_ = receiver_env_;
  using operation_state_base_ = manual_child_operation_state<
    operation_state,
    env_,
    Sender&>;
  template<typename Child>
  class child_;
  // ...
};
```

LSEG

Operation State

This environment is wrong because it doesn't provide an association with our chained stop source.

```
template<typename Sender, typename Receiver>
class operation_state :
  inlinable_operation_state<
    operation_state<Sender, Receiver>,
    Receiver>,
  public manual_child_operation_state<
    operation_state<Sender, Receiver>,
    std::env_of_t<Receiver>,
    Sender&>
{
  using receiver_base_ = inlinable_operation_state<
    operation_state,
    Receiver>;
  using receiver_base_::get_receiver;
  using receiver_env_ = std::env_of_t<Receiver>;
  using env_ = receiver_env_;
  using operation_state_base_ = manual_child_operation_state<
    operation_state,
    env_,
    Sender&>;
  template<typename Child>
  class child_;
  // ...
};
```

LSEG

Operation State

Fixed.

`stop_source_env` was introduced/covered in the C++Now 2025 talk.

```
template<typename Sender, typename Receiver>
class operation_state :
  inlinable_operation_state<
    operation_state<Sender, Receiver>,
    Receiver>,
  public manual_child_operation_state<
    operation_state<Sender, Receiver>,
    stop_source_env<std::env_of_t<Receiver>>,
    Sender&>
{
  using receiver_base_ = inlinable_operation_state<
    operation_state,
    Receiver>;
  using receiver_base_::get_receiver;
  using receiver_env_ = std::env_of_t<Receiver>;
  using env_ = stop_source_env<receiver_env_>;
  using operation_state_base_ = manual_child_operation_state<
    operation_state,
    env_,
    Sender&>;
  template<typename Child>
  class child_;
  // ...
};
```

LSEG

Operation State

Member variables.

```
template<typename Sender, typename Receiver>
class operation_state : /* ... */ {
  // ...
  Sender s_;
  chained_stop_source<
    std::execution::stop_token_of_t<receiver_env_>> source_;
  std::atomic<std::size_t> outstanding_{1};
  std::atomic<bool> arrived_{false};
  storage_for_completion_signatures<
    typename [:completion_signatures(
      dealias(
        ^^std::execution::completion_signatures_of_t<
          Sender&,
          env_>),
      dealias(^^env_)):]> storage_;
  /* ...
   *
   *
   *
   *
   *
   *
   *
   */
};
```

LSEG

Operation State

If there are outstanding operations finalization does not proceed.

Otherwise finalization begins with detaching from the stop token.

```
template<typename Sender, typename Receiver>
class operation_state : /* ... */ {
  // ...
  constexpr void maybe_complete_() noexcept {
    if (outstanding_.fetch_sub(
      1,
      std::memory_order_acq_rel) != 1)
    {
      return;
    }
    source_.detach();
    /* ...
     *
     *
     */
  }
  /* ...
   *
   *
   *
   *
   *
   *
   */
};
```

LSEG

Operation State

If an error or stopped completion signature is stored the overall operation completes therewith.

```
template<typename Sender, typename Receiver>
class operation_state : /* ... */ {
  // ...
  constexpr void maybe_complete_() noexcept {
    // ...
    if (std::move(storage_).visit([&](
      const auto& tag, auto&&... args) noexcept
    {
      tag(
        std::move(get_receiver()),
        std::forward<decltype(args)>(args)...);
    })) {
      return;
    }
    // ...
  }
  /* ...
   *
   *
   *
   *
   *
   *
   */
};
```

LSEG

Operation State

It must be the case that the main sender sent nullary `set_value` and therefore we send that.

Note the `if constexpr` since it may be the case that the main sender can't send nullary `set_value` and in that situation the guarded branch isn't guaranteed to compile.

```
template<typename Sender, typename Receiver>
class operation_state : /* ... */ {
  // ...
  constexpr void maybe_complete_() noexcept {
    /* ...
     *
     *
     */
    if constexpr (requires {
      storage_.arrive(std::execution::set_value);
    }) {
      std::execution::set_value(std::move(get_receiver()));
      return;
    }
    std::unreachable();
  }
  /* ...
   *
   *
   *
   *
   *
   *
   */
};
```

LSEG

Operation State

Stores the first non-successful completion from the main or any child operation and requests that all outstanding operations end as soon as possible (via a stop request).

```
template<typename Sender, typename Receiver>
class operation_state : /* ... */ {
  // ...
  template<typename Tag, typename... Args>
  constexpr void store_(const Tag& tag, Args&&... args) noexcept {
    if (arrived_.exchange(true, std::memory_order_relaxed)) {
      return;
    }
    source_.request_stop();
    try {
      storage_.arrive(tag, std::forward<Args>(args)...);
    } catch (...) {
      storage_.arrive(
        std::execution::set_error,
        std::current_exception());
    }
  }
  /* ...
   *
   *
   *
   *
   */
};
```

LSEG

Operation State

Starts the next iteration of the main sender.

```
template<typename Sender, typename Receiver>
class operation_state : /* ... */ {
  // ...
  constexpr void start_() noexcept {
    try {
      operation_state_base_::construct(s_);
    } catch (...) {
      store_(
        std::execution::set_error,
        std::current_exception());
      maybe_complete_();
      return;
    }
    operation_state_base_::start();
  }
  /* ...
   *
   *
   *
   *
   *
   *
   *
   */
};
```

LSEG

Operation State

`impl` creates and starts a child sender.

`get_allocator_for` is from C++Now 2025 talk.

```
template<typename Sender, typename Receiver>
class operation_state : /* ... */ {
  // ...
  template<std::execution::sender Child>
  constexpr bool spawn_(Child&& sender) noexcept {
    auto alloc = ::get_allocator_for<child_<Child>>(
      std::get_env(get_receiver()));
    using traits = std::allocator_traits<decltype(alloc)>;
    const auto impl = [&]() {
      const auto ptr = traits::allocate(alloc, 1);
      try {
        traits::construct(
          alloc, ptr, *this, std::forward<Child>(sender));
      } catch (...) {
        traits::deallocate(alloc, ptr, 1);
        throw;
      }
      outstanding_.fetch_add(1, std::memory_order_relaxed);
      ptr->start();
    };
    // ...
  }
  // ...
};
```

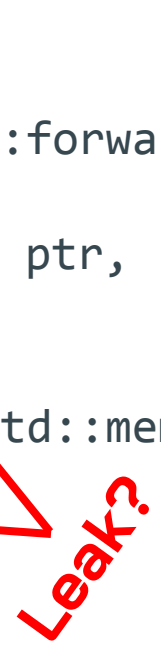
LSEG

Operation State

Is this pointer leaked?

No, thanks to structured concurrency we can rely on the child completing once `start` is called.

```
template<typename Sender, typename Receiver>
class operation_state : /* ... */ {
  // ...
  template<std::execution::sender Child>
  constexpr bool spawn_(Child&& sender) noexcept {
    auto alloc = ::get_allocator_for<child_<Child>>(
      std::get_env(get_receiver()));
    using traits = std::allocator_traits<decltype(alloc)>;
    const auto impl = [&]() {
      const auto ptr = traits::allocate(alloc, 1);
      try {
        traits::construct(
          alloc, ptr, *this, std::forward<Child>(sender));
      } catch (...) {
        traits::deallocate(alloc, ptr, 1);
        throw;
      }
      outstanding_.fetch_add(1, std::memory_order_relaxed);
      ptr->start();
    };
    // ...
  }
  // ...
};
```



LSEG

Operation State

Coalesce exceptions to fan in and failure of the overall operation.

```
template<typename Sender, typename Receiver>
class operation_state : /* ... */ {
  // ...
  template<std::execution::sender Child>
  constexpr bool spawn_(Child&& sender) noexcept {
    /* ...
     *
     *
     *
     *
     */
    try {
      impl();
    } catch (...) {
      store_(
        std::execution::set_error,
        std::current_exception());
      return false;
    }
    return true;
  }
  /* ...
   */
};
```

LSEG

Operation State

Opt in concept and constructor.

```
template<typename Sender, typename Receiver>
class operation_state : /* ... */ {
  // ...
public:
  using operation_state_concept =
    std::execution::operation_state_t;
  constexpr explicit operation_state(Sender s, Receiver r)
    : receiver_base_(std::move(r)),
      s_(std::move(s))
  {}
  /* ...
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   */
};
```

LSEG

Operation State

Attach to the stop token provided by the receiver's environment and start the main loop.

```
template<typename Sender, typename Receiver>
class operation_state : /* ... */ {
  // ...
  constexpr void start() & noexcept {
    try {
      source_.attach(
        std::execution::get_stop_token(
          std::get_env(get_receiver())));
    } catch (...) {
      std::execution::set_error(
        std::move(get_receiver()),
        std::current_exception());
      return;
    }
    start_();
  }
  /* ...
   *
   *
   *
   *
   *
   *
   */
};
```

LSEG

Operation State

Allows children to obtain an environment associated with our stop source.

```
template<typename Sender, typename Receiver>
class operation_state : /* ... */ {
  // ...
  constexpr env_ get_env() noexcept {
    return env_(source_, std::get_env(get_receiver()));
  }
  /* ...
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   */
};
```

LSEG

Operation State

If the main sender completes with no arguments it’s done and therefore we begin graceful fan in (i.e. we don’t cancel children and allow all spawned work to complete).

```
template<typename Sender, typename Receiver>
class operation_state : /* ... */ {
  // ...
  constexpr void set_value() noexcept {
    operation_state_base_::destruct();
    maybe_complete_();
  }
  /* ...
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   */
};
```

LSEG

Operation State

If the main sender completes with arguments those arguments are senders so we fan out to those operations.

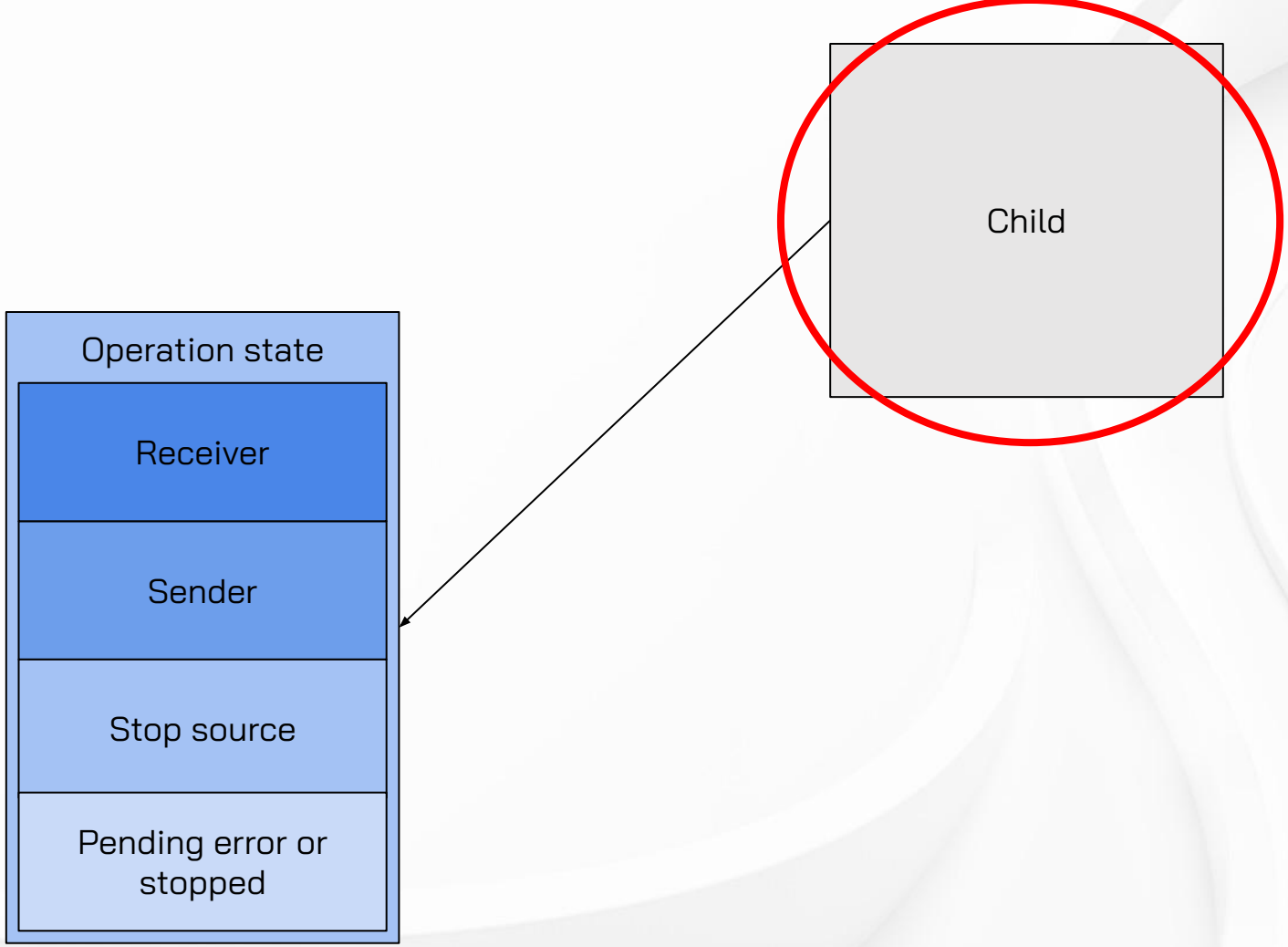
```
template<typename Sender, typename Receiver>
class operation_state : /* ... */ {
  // ...
  template<typename... Args>
  constexpr void set_value(Args&&... args) noexcept {
    const auto success = (spawn_(std::forward<Args>(args)) && ...);
    operation_state_base_::destruct();
    if (success) {
      start_();
    } else {
      maybe_complete_();
    }
  }
  /* ...
   *
   *
   *
   *
   *
   *
   *
   *
   *
   */
};
```

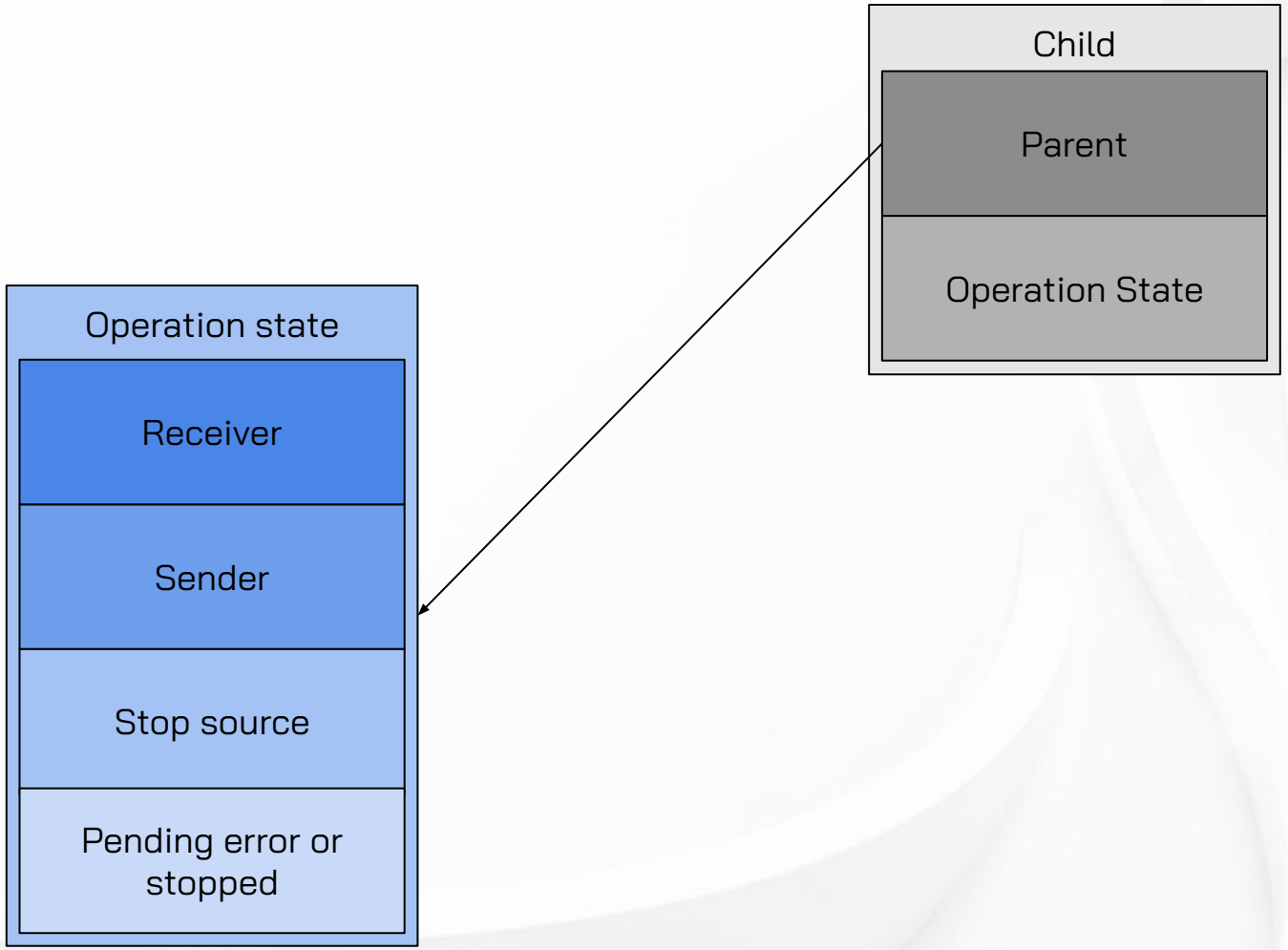
LSEG

Operation State

Error and stopped from the main sender trigger fan in.

```
template<typename Sender, typename Receiver>
class operation_state : /* ... */ {
  // ...
  template<typename... Args>
  constexpr void set_error(Args&&... args) noexcept {
    store_(
      std::execution::set_error,
      std::forward<Args>(args)...);
    operation_state_base_::destruct();
    maybe_complete_();
  }
  template<typename... Args>
  constexpr void set_stopped(Args&&... args) noexcept {
    store_(
      std::execution::set_stopped,
      std::forward<Args>(args)...);
    operation_state_base_::destruct();
    maybe_complete_();
  }
};
```


Child
Operation state
Receiver
Sender
Stop source
Pending error or stopped


Child
Parent
Operation State
Operation state
Receiver
Sender
Stop source
Pending error or stopped

Child

Boilerplate.

Note that `child_operation_state` doesn't have manual lifetime management (constructing connects).

```
template<typename Sender, typename Receiver>
template<typename Child>
class operation_state<Sender, Receiver>::child_ :
  public manual_child_operation_state<
    child_<Child>,
    env_,
    Child>
{
  using base_ = manual_child_operation_state<
    child_<Child>,
    env_,
    Child>;
  operation_state& self_;
  /* ...
   *
   *
   *
   *
   *
   *
   *
   *
   *
   */
};
```

LSEG

Child

Destroys `*this` and reports completion back to the main operation state.

Structured concurrency allows us to rely on this running (we're always going to receive a completion signal because that's implied by the receiver contract).

```
template<typename Sender, typename Receiver>
template<typename Child>
class operation_state<Sender, Receiver>::child_ : /* ... */ {
  // ...
  constexpr void complete_() noexcept {
    base_::destruct();
    auto alloc = ::get_allocator_for<child_>(
      std::get_env(self_.get_receiver()));
    using traits = std::allocator_traits<decltype(alloc)>;
    auto&& op = self_;
    traits::destroy(alloc, this);
    traits::deallocate(alloc, this, 1);
    op.maybe_complete_();
  }
  /* ...
   *
   *
   *
   *
   *
   *
   *
   *
   */
};
```

LSEG

Child

Stores error or stopped in the main operation state (unless such has already been reported) and then fans this child in.

```
template<typename Sender, typename Receiver>
template<typename Child>
class operation_state<Sender, Receiver>::child_ : /* ... */ {
  // ...
  template<typename Tag, typename... Args>
  constexpr void fail_(const Tag& tag, Args&&... args) noexcept {
    self_.store_(tag, std::forward<Args>(args)...);
    complete_();
  }
  /* ...
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   */
};
```

LSEG

Child

Boilerplate.

```
template<typename Sender, typename Receiver>
template<typename Child>
class operation_state<Sender, Receiver>::child_ : /* ... */ {
  // ...
public:
  constexpr explicit child_(
    operation_state& self,
    Child&& child)
      : self_(self)
  {
    base_::construct(std::forward<Child>(child));
  }
  using base_::start;
  /* ...
   *
   *
   *
   *
   *
   *
   *
   *
   *
   */
};
```

LSEG

Child

Provides the interface expected by the `child_operation_state` CRTP base class.

```
template<typename Sender, typename Receiver>
template<typename Child>
class operation_state<Sender, Receiver>::child_ : /* ... */ {
  // ...
  constexpr env_ get_env() noexcept {
    return self_.get_env();
  }
  template<typename... Args>
  constexpr void set_value(Args&&...) noexcept {
    complete_();
  }
  template<typename... Args>
  constexpr void set_error(Args&&... args) noexcept {
    fail_(
      std::execution::set_error,
      std::forward<Args>(args)...);
  }
  template<typename... Args>
  constexpr void set_stopped(Args&&... args) noexcept {
    fail_(
      std::execution::set_stopped,
      std::forward<Args>(args)...);
  }
};
```

LSEG

Asynchronous Network Server

Asynchronous code from earlier.

```
std::execution::sender auto accept(int fd);

std::execution::sender auto handle_connection(int fd);

const int accepting = /* ... */;
std::execution::sender auto server =
  accept(accepting) |
  std::execution::let_value(handle_connection) |
  repeat();
```

LSEG

Asynchronous Network Server

The first two stages form a higher order sender (an asynchronous operation which sends a sender) which is then piped into `branch` for dynamic fan out.

```
std::execution::sender auto accept(int fd);

std::execution::sender auto handle_connection(int fd);

const int accepting = /* ... */;
std::execution::sender auto server =
  accept(accepting) |
  std::execution::then([](const int fd) {
    return handle_connection(fd);
  }) |
  branch();
```

LSEG

Summary

Dynamic Asynchronous Graph Execution

- Primitives provided out of the box by `std::execution` don't permit dynamic execution graphs, or graphs with cycles
  - Real world programs commonly feature both
  - Both can be added to `std::execution`
- `std::execution` is not a closed set of algorithms
  - It's a *lingua franca* for asynchronous computation in C++
  - The intention is that it will be extended (like the STL before it)
- `repeat` and `branch` are just two (of an infinite set) of the algorithms with which `std::execution` can be enriched

Thank you

Robert Leahy
Lead Software Engineer
rleahy@rleahy.ca

LSEG

Questions?

LSEG