Consuming Senders from Coroutine-Native Code

Document Number:	P4092R1
Date:	2026-05-01
Intent:	Inform
Audience:	LEWG

Reply-to: Vinnie Falco vinnie.falco@gmail.com
Steve Gerbino steve@gerbino.co
C++ Alliance Proposal Team

Table of Contents

Abstract

A single class template bridges sender-based code into coroutine-native I/O with inline operation state, correct stop propagation, and automatic dispatch-back.

An `IoAwaitable` bridge (P4003R3 [1]) consumes `std::execution` (P2300R10 [2]) senders with inline operation state, correct stop token propagation, and automatic executor dispatch-back. The bridge is one class template. The complete implementation is in Appendix A.

Revision History

R1: May 2026 (pre-Brno mailing)

- Formatting corrections.

R0: April 2026 (post-Croydon mailing)

- Initial version.

---

1. Disclosure

The author provides information and serves at the pleasure of the committee.

This paper is part of the Network Endeavor (P4100R0), a project to bring coroutine-native I/O to C++.

The author developed and maintains Capy [3] and Corosio [4] and believes coroutine-native I/O is a practical foundation for networking in C++.

Coroutine-native I/O and `std::execution` are complementary. Each serves the domain where its design choices pay off.

The bridge depends on Capy [3] (coroutine primitives, no sockets, no platform I/O) and `beman::execution` [5].

This paper asks for nothing.

---

2. The Bridge

`await_sender` consumes a `std::execution` sender from inside a coroutine. The sender runs on whatever scheduler it was given; the coroutine resumes on its own executor.

```
capy::task<int> compute(auto sched)
{
    int result = co_await capy::await_sender(
        ex::schedule(sched)
            | ex::then([] {
                std::cout
                    << "  sender running on thread "
                    << std::this_thread::get_id()
                    << "\n";
                return 42 * 42;
            }));

    std::cout
        << "  coroutine resumed on thread "
        << std::this_thread::get_id() << "\n";

    co_return result;
}
```

`await_sender` returns a `sender_awaitable` satisfying `IoAwaitable` (P4003R3 [1]). Any coroutine type that propagates `io_env` through `await_suspend(h, io_env const*)` can use it. The two-argument form is deliberate: the compiler rejects any coroutine type that does not propagate `io_env`, enforcing the sandbox boundary at compile time. Complete implementation in Appendix A.

---

3. Demonstration

Output from the example in Section 2, compiled with MSVC 19.43 against Capy [3] and `beman::execution` [5] (a community implementation of `std::execution`):

```
main thread: 32208
  sender running on thread 9560
  coroutine resumed on thread 34356
result: 1764
```

The sender ran on the `run_loop` thread. The coroutine resumed on the Capy thread pool. Zero bytes allocated beyond the coroutine frame.

---

4. What the Bridge Does

The bridge consumes any `std::execution` sender whose value completion signature is a single type or `void`. Operation state stored inline. `set_value`, `set_error`, `set_stopped` handled. Standard pipelines - `when_all`, `then`, `let_value`, `on` - compose with the bridge.

The bridge inspects error completion signatures at compile time. If the sender advertises `set_error(std::error_code)`, `await_resume` returns `io_result<T>`:

```
auto [ec, val] = co_await await_sender(sndr);
```

No exceptions for `error_code`. Otherwise `await_resume` returns `T` directly; genuine exceptions are rethrown and cancellation is surfaced as an exception. Static dispatch, zero runtime cost. The `operation_cancelled` type in Appendix A is illustrative; a production implementation would use a project-appropriate cancellation exception.

The error-code dispatch is the consuming side of the **abstraction floor** (P4093R0 [6] Section 4):

Region	What the code sees
Above the floor	`error_code` alone - composition works
Below the floor	`(error_code, size_t)` - both values intact

When the sender completes, the bridge posts the resumption back to the coroutine's originating executor. The coroutine resumes in the correct context regardless of where the sender executed.

Does not use `execution::task`.

---

5. What the Bridge Does Not Require

`std::execution::task` (P3552R3 [7], revised at Croydon by P3980R1 [8], P3941R4 [9], P3927R2 [10], P4151R1 [11]) is a sender-returning coroutine type. Its promise type-erases the downstream receiver, and its error path converts `error_code` to `exception_ptr` via `AS-EXCEPT-PTR`. None of the Croydon revisions changed these properties. The bridge avoids both mechanisms:

Property	`execution::task`	Bridge
Routine I/O errors become exceptions	Yes (via `AS-EXCEPT-PTR`)	No
Type-erased dispatch	Yes (promise continuation)	No
Allocation beyond coroutine frame	Yes (inner coroutine frame)	No
Requires `std::execution::task`	-	No

The bridge consumes senders without `std::execution::task`.

---

6. The Narrowest Abstraction

The bridge depends on two things: Capy [3] (coroutine primitives) and `std::execution` (sender/receiver protocol). No platform I/O. No networking headers. No additional coroutine type. The implementation in Appendix A uses `beman::execution` [5], a community implementation of `std::execution`, but the bridge requires only the standard sender/receiver concepts.

Narrower than `execution::task`: the bridge does not type-erase, does not allocate, and does not impose an `Environment` parameter. Narrower than a completion-token adapter: the bridge does not require Asio or any I/O service. Narrower than `with_awaitable_senders` (P2300R10 [2] Section 4.17): the bridge uses the IoAwaitable two-argument `await_suspend` to receive the environment, uses `post()` for guaranteed executor dispatch-back rather than relying on scheduler affinity or atomic synchronization for synchronous completions, and separates `error_code` from `exception_ptr` at compile time rather than converting through `unhandled_stopped()`. The coroutine type the programmer already uses does not change when the bridge is added.

The bridge is the proof that coexistence works. Senders compose. Coroutines do I/O. One class template connects them.

---

7. Acknowledgments

The authors thank Dietmar Kühl for `beman::execution` [5] and for the channel-routing enumeration in P2762R2 [12], Michał Dominiak, Eric Niebler, and Lewis Baker for `std::execution`, Chris Kohlhoff for identifying the partial-success problem in P2430R0 [13], Kirk Shoop for the completion-token heuristic analysis in P2471R1 [14], Fabio Fracassi for P3570R2 [15], Ville Voutilainen and Jens Maurer for reflector discussion on dispatch patterns, Herb Sutter for identifying the need for tutorials and documentation, Mark Hoemmen for insights on `std::linalg` and the layered abstraction model, and Peter Dimov for the refined channel mapping.

References

[1] P4003R3 - "A Minimal Coroutine Execution Model" (Vinnie Falco, Steve Gerbino, Mungo Gill, 2026).

[2] P2300R10 - "std::execution" (Michał Dominiak et al., 2024).

[3] cppalliance/capy - Coroutine primitives library.

[4] cppalliance/corosio - Coroutine-native networking library.

[5] bemanproject/execution - Community implementation of `std::execution`.

[6] P4093R0 - "Producing Senders from Coroutine-Native Code" (Vinnie Falco, Steve Gerbino, 2026).

[7] P3552R3 - "Add a Coroutine Task Type" (Dietmar Kühl, Maikel Nadolski, 2025).

[8] P3980R1 - "Task's Allocator Use" (Dietmar Kühl, 2026).

[9] P3941R4 - "Scheduler Affinity" (Dietmar Kühl, 2026).

[10] P3927R2 - "`task_scheduler` Bulk Execution" (Eric Niebler, 2026).

[11] P4151R1 - "Rename `affine_on`" (Robert Leahy, 2026).

[12] P2762R2 - "Sender/Receiver Interface For Networking" (Dietmar Kühl, 2023).

[13] P2430R0 - "Slides: Partial success scenarios with P2300" (Chris Kohlhoff, 2021).

[14] P2471R1 - "NetTS, ASIO and Sender Library Design Comparison" (Kirk Shoop, 2021).

[15] P3570R2 - "Optional variants in sender/receiver" (Fabio Fracassi, 2025).

Appendix A. Bridge Implementation

```
#include <boost/capy/error.hpp>
#include <boost/capy/ex/io_env.hpp>
#include <boost/capy/io_result.hpp>

#include <beman/execution/execution.hpp>

#include <coroutine>
#include <exception>
#include <new>
#include <stop_token>
#include <system_error>
#include <tuple>
#include <type_traits>
#include <utility>
#include <variant>

namespace boost::capy {

namespace detail {

struct stopped_t {};

struct operation_cancelled {};

struct bridge_env
{
    std::stop_token st_;

    auto query(
        beman::execution::get_stop_token_t const&)
            const noexcept
    {
        return st_;
    }
};

template<class Sender>
using sender_single_value_t =
    beman::execution::value_types_of_t<
        Sender,
        bridge_env,
        std::tuple,
        std::type_identity_t>;
```

```
template<class Sender>
struct has_error_code_completion
{
    template<class... Es>
    struct checker
    {
        static constexpr bool value =
            (std::is_same_v<
                Es, std::error_code> || ...);
    };

    static constexpr bool value =
        beman::execution::error_types_of_t<
            Sender,
            bridge_env,
            checker>::value;
};

template<class Sender>
constexpr bool has_error_code_v =
    has_error_code_completion<Sender>::value;

// Variant when sender can complete with
// set_error(error_code): separate slot for
// error_code so it is not wrapped in
// exception_ptr.
template<class ValueTuple>
using ec_result_variant = std::variant<
    std::monostate,
    ValueTuple,
    std::error_code,
    std::exception_ptr,
    stopped_t>;

// Variant when sender does not complete with
// set_error(error_code).
template<class ValueTuple>
using no_ec_result_variant = std::variant<
    std::monostate,
    ValueTuple,
    std::exception_ptr,
    stopped_t>;

template<class ValueTuple, bool HasEc>
```

```
using result_variant = std::conditional_t<
    HasEc,
    ec_result_variant<ValueTuple>,
    no_ec_result_variant<ValueTuple>>;

template<class ValueTuple, bool HasEc>
struct bridge_receiver
{
    using receiver_concept =
        beman::execution::receiver_t;

    result_variant<ValueTuple, HasEc>* result_;
    std::coroutine_handle<>            cont_;
    io_env const*                      env_;

    auto get_env() const noexcept -> bridge_env
    {
        return {env_->stop_token};
    }

    template<class... Args>
    void set_value(Args&&... args) && noexcept
    {
        result_->template emplace<1>(
            std::forward<Args>(args)...);
        env_->executor.post(cont_);
    }

    template<class E>
    void set_error(E&& e) && noexcept
    {
        if constexpr (
            HasEc &&
            std::is_same_v<
                std::decay_t<E>,
                std::error_code>)
            result_->template emplace<2>(
                std::forward<E>(e));
        else if constexpr (
            std::is_same_v<
                std::decay_t<E>,
                std::exception_ptr>)
        {
            constexpr auto idx = HasEc ? 3 : 2;
            result_->template emplace<idx>(
```

```
                std::forward<E>(e));
        }
        else
        {
            constexpr auto idx = HasEc ? 3 : 2;
            result_->template emplace<idx>(
                std::make_exception_ptr(
                    std::forward<E>(e)));
        }
        env_->executor.post(cont_);
    }

    void set_stopped() && noexcept
    {
        constexpr auto idx = HasEc ? 4 : 3;
        result_->template emplace<idx>(
            stopped_t{});
        env_->executor.post(cont_);
    }
};

} // namespace detail

template<class Sender>
struct sender_awaitable
{
    static constexpr bool has_ec =
        detail::has_error_code_v<Sender>;

    using value_tuple =
        detail::sender_single_value_t<Sender>;
    using variant_type =
        detail::result_variant<
            value_tuple, has_ec>;
    using receiver_type =
        detail::bridge_receiver<
            value_tuple, has_ec>;
    using op_state_type = decltype(
        beman::execution::connect(
            std::declval<Sender>(),
            std::declval<receiver_type>()));

    Sender sndr_;
    variant_type result_{};
```

```
alignas(op_state_type)
unsigned char op_buf_[sizeof(op_state_type)];
bool op_constructed_ = false;

explicit sender_awaitable(Sender sndr)
    : sndr_(std::move(sndr))
{
}

sender_awaitable(sender_awaitable&& o)
    noexcept(
        std::is_nothrow_move_constructible_v<
            Sender>)
    : sndr_(std::move(o.sndr_))
{
}

sender_awaitable(
    sender_awaitable const&) = delete;
sender_awaitable& operator=(
    sender_awaitable const&) = delete;
sender_awaitable& operator=(
    sender_awaitable&&) = delete;

~sender_awaitable()
{
    if(op_constructed_)
        std::launder(
            reinterpret_cast<op_state_type*>(
                op_buf_))->~op_state_type();
}

bool await_ready() const noexcept
{
    return false;
}

std::coroutine_handle<>
await_suspend(
    std::coroutine_handle<> h,
    io_env const* env)
{
    ::new(op_buf_) op_state_type(
        beman::execution::connect(
            std::move(sndr_),
```

```
                receiver_type{
                    &result_, h, env}));
        op_constructed_ = true;
        beman::execution::start(
            *std::launder(
                reinterpret_cast<
                    op_state_type*>(
                        op_buf_)));
        return std::noop_coroutine();
    }

    auto await_resume()
    {
        if constexpr (has_ec)
            return await_resume_ec();
        else
            return await_resume_no_ec();
    }

private:
    // Sender can complete with
    // set_error(error_code). Return io_result
    // so the error code is a value, not an
    // exception.
    auto await_resume_ec()
    {
        // exception_ptr at index 3
        if(result_.index() == 3)
            std::rethrow_exception(
                std::get<3>(result_));

        if constexpr (
            std::tuple_size_v<
                value_tuple> == 0)
        {
            // stopped at index 4
            if(result_.index() == 4)
                return io_result<>{
                    make_error_code(
                        error::canceled)};
            if(result_.index() == 2)
                return io_result<>{
                    std::get<2>(result_)};
            return io_result<>{};
        }
```

```
    else if constexpr (
        std::tuple_size_v<
            value_tuple> == 1)
    {
        using T = std::tuple_element_t<
            0, value_tuple>;
        if(result_.index() == 4)
            return io_result<T>{
                make_error_code(
                    error::canceled)};
        if(result_.index() == 2)
            return io_result<T>{
                std::get<2>(result_)};
        return io_result<T>{
            {},
            std::get<0>(
                std::get<1>(
                    std::move(result_)))};
    }
    else
    {
        if(result_.index() == 4)
            return io_result<value_tuple>{
                make_error_code(
                    error::canceled)};
        if(result_.index() == 2)
            return io_result<value_tuple>{
                std::get<2>(result_)};
        return io_result<value_tuple>{
            {},
            std::get<1>(
                std::move(result_))};
    }
}

// Sender does not complete with
// set_error(error_code). Return the value
// directly; rethrow exceptions.
auto await_resume_no_ec()
{
    // exception_ptr at index 2
    if(result_.index() == 2)
        std::rethrow_exception(
            std::get<2>(result_));
    // stopped at index 3
```

```
        if(result_.index() == 3)
            throw detail::operation_cancelled{};

        if constexpr (
            std::tuple_size_v<
                value_tuple> == 0)
            return;
        else if constexpr (
            std::tuple_size_v<
                value_tuple> == 1)
            return std::get<0>(
                std::get<1>(
                    std::move(result_)));
        else
            return std::get<1>(
                std::move(result_));
    }
};

template<class Sender>
auto await_sender(Sender&& sndr)
{
    return sender_awaitable<
        std::decay_t<Sender>>(
            std::forward<Sender>(sndr));
}

} // namespace boost::capy
```