Trade-offs in Asynchronous Abstraction Design

Document Number:	P4178R0
Date:	2026-05-01
Intent:	Inform
Audience:	WG21

Reply-to:	Vinnie Falco vinnie.falco@gmail.com

Table of Contents

Abstract

Sections of P2300R10 are compared to other sections of P2300R10 and entered into the historical record.

This paper reviews P2300R10 for internal consistency using only the specification's own text. Each finding pairs two passages from the same paper that appear to be in tension. Four recurring patterns organize the findings. Where a charitable reading resolves the tension, it is noted. No external sources are cited.

Revision History

R0: May 2026 (pre-Brno mailing)

- Initial version.

---

1. Disclosure

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

The author developed and maintains Capy and Corosio 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.

This paper asks for nothing.

---

2. Four Patterns

Pattern	Description
1. Yes - But - Actually	Claim walked back within the same section
2. This Way, Then That Way	Different evidentiary standards for the same concern
3. Dismiss, Then Adopt	Prior art condemned, then used as foundation
4. First Yes, Then No	Stated priorities contradicted by the design

---

3. The Record

Yes - But - Actually

Four consecutive sections about the coroutine/sender interface follow the same structure.

	P2300R10 Section 4.14
Yes	"it is sensible to pass both the error code and the result (if any) through the value channel"
But	"In other cases, the partial success is more of a partial failure."
Actually	"sending the error through the value channel loses valuable contextual information"

The section may be exploring a genuine design tension rather than contradicting itself - partial success genuinely admits two reasonable framings.

	P2300R10 Section 4.16
Yes	"Many senders can be trivially made awaitable"
But	"If the coroutine type opts in to sender support"
Actually	"Only some senders can be made awaitable directly because of the fact that callbacks are more expressive than coroutines."

The progression from "many" to "only some" may reflect an honest narrowing of scope as the section develops the details.

	P2300R10 Section 4.15
Yes	"In truth there will be no problem"
But	"By 'generally awaitable' we mean types that don't require custom `await_transform` trickery"
Actually	"If you are not bothered by the possibility of allocations and indirections"

The intended audience for Section 4.15 may be users who have already accepted these costs for the convenience of general awaitability. Section 1.9.2 calls these same allocations and indirections a "deal-breaker."

	P2300R10 Section 4.17
Yes	"Cancellation of a sender can unwind a stack of coroutines"
But	"Provided that the promise types of the calling coroutines also inherit from `with_awaitable_senders`"
Actually	"Many promise types will not implement `unhandled_stopped`. When an uncatchable stopped exception tries to propagate through such a coroutine, it is treated as an unhandled exception and `terminate` is called."

The capability is real for promise types that opt in. The failure mode affects promise types that do not, and the section documents a limitation honestly rather than hiding it.

This Way, Then That Way

P2300R10 Section 1.9.2 (paragraph 2)	P2300R10 Section 1.9.2 (paragraph 4)
"coroutine frames require an unavoidable dynamic allocation"	"the HALO optimization...completely eliminating the runtime overhead"

The authors may view HALO as too unreliable to change the fundamental cost model, even while acknowledging its existence as a compiler optimization.

P2300R10 Section 1.9.2	P2300R10 Section 4.9.4
"the extra allocations and indirections are a deal-breaker"	"concerns related to runtime overhead of the design in single-threaded scenarios and these concerns are still being investigated"

Coroutine allocation costs are well-characterized from years of field deployment. Sender overhead in single-threaded scenarios is a newer area of study. The different treatment may reflect different levels of maturity.

P2300R10 Section 1.9.2	P2300R10 Section 5.6
"HALO requires a sophisticated compiler, and a fair number of stars need to align"	"the compiler is able to see a chain of work described using senders as a tree of tail calls, allowing for inlining and removal of most of the sender machinery"

Sender optimization relies on inlining statically dispatched calls - a simpler class of optimization than the alias analysis HALO requires. The different confidence levels may reflect this structural difference.

P2300R10 Section 1.9.2 (conclusion)	P2300R10 Section 1.9.2 (sole evidence)
"the extra allocations and indirections are a deal-breaker"	"In our experience, more often than not"

The authors' extensive production experience - documented in Section 1.10.1 - may justify stronger conclusions than "more often than not" would normally support.

Dismiss, Then Adopt

P2300R10 Section 1.9.3	The receiver concept
"The lack of a standard callback shape obstructs generic design"	`set_value`, `set_error`, `set_stopped` - a standardized callback shape with cancellation support

Defining a standard callback shape is arguably the thesis of the receiver concept, not a contradiction of the prior-art critique.

P2300R10 Section 1.9.1	P2300R10 Section 4.8
"Futures, as traditionally realized, require the dynamic allocation and management of a shared state, synchronization, and typically type-erasure of work and continuation... These expenses rule out the future abstraction for many uses"	`split` converts single-shot to multi-shot using shared state, synchronization, and dynamic allocation

Unlike futures, where these costs are mandatory, `split` makes them opt-in. Users who do not fork do not pay.

P2300R10 Section 1.9.2	P2300R10 Section 1.10.1
"we consider coroutines a poor choice for a basis of all standard async"	"In all the cases mentioned above, developers mix-and-match between the sender algorithms in Unifex and Unifex's coroutine type, `unifex::task`."

This mix-and-match usage may reflect the intended design - senders as the composable foundation, coroutines as the ergonomic layer.

P2300R10 Section 1.9.2	P2300R10 Section 1.10.1
"we consider coroutines a poor choice for a basis of all standard async"	"Coroutines are easier to understand than chained futures so the team was able to meet requirements for certain constrained environments that would have been too complicated to maintain with futures."

The authors likely intend a distinction between coroutines as a basis abstraction and coroutines as a user-facing tool. Production use of coroutines alongside senders is consistent with this layered view.

P2300R10 Section 1.9.2	P2300R10 Section 4.15
"we consider coroutines a poor choice for a basis of all standard async"	"we expect that coroutines and awaitables will be how a great many will choose to express their asynchronous code"

If coroutines are understood as a consumer of the sender abstraction rather than a competitor to it, this relationship is architectural rather than contradictory.

First Yes, Then No

P2300R10 Section 4.9.4	Context
"these concerns are still being investigated" / "This paper will be updated in the future with any changes that arise from the investigations into P2175R0."	This is R10, the revision that became C++26.

All shipped standards contain open questions. The committee may have judged the specification ready despite these items, with investigation continuing in parallel.

P2300R10 Section 1.2	P2300R10 Section 4.20.11
"Have clear and concise answers for where things execute."	"It completes inline on the execution resource on which the last input sender completes"

The answer may be clear at a different level of abstraction - the last child to complete determines the execution context, which is a well-defined rule even if the result varies at runtime.

P2300R10 Section 1.2	P2300R10 Section 4.17
"Support cancellation, which is not an error."	"the coroutine behaves as if an uncatchable exception had been thrown from the `co_await` expression"

Implementing cancellation through an exception-like mechanism in coroutines may be the least-bad option given the constraints of the coroutine specification, even if it blurs the conceptual boundary.

P2300R10 Section 1.2	P2300R10 Section 4.9.3
"Make it easy to be correct by construction."	"A stop-request can be issued concurrently from another thread. This means the implementation of `execution::start()` needs to be careful to ensure that, once a stop callback has been registered, that there are no data-races"

Cancellation in concurrent systems is inherently difficult. The tension between correctness by construction and the practical complexity of stop-request handling may be an honest acknowledgment of a hard problem rather than a design failure.

P2300R10 Section 4.8	P2300R10 Section 4.7
"Senders are forkable"	"A single-shot sender can only be connected to a receiver at most once."

The design may intend single-shot as the zero-overhead default, with `split` as an explicit opt-in for use cases that genuinely require shared state.

---

4. Example Code

The following code is from P2300R10. No external code is shown.

The `retry` algorithm

Section 4.16 presents a coroutine implementation of `retry` and calls it "trivial":

```
template<class S>
  requires single-sender<S&>
task<single-sender-value-type<S>> retry(S s) {
  for (;;) {
    try {
      co_return co_await s;
    } catch(...) {
    }
  }
}
```

Section 1.5.2 presents the sender implementation of the same algorithm. It requires a custom receiver, an operation state with optional re-emplacement, a conversion utility for non-movable types, a sender with completion signature transforms, and exception handling across retry boundaries. The full implementation is approximately 125 lines.

The sender implementation provides compile-time composition and type safety that the coroutine version does not - library authors write this machinery once so that users can write the seven-line version. This is the `_retry_receiver`:

```
template<class S, class R>
struct _retry_receiver {
  _retry_op<S, R>* o_;

  void set_value(auto&&... as) && noexcept {
    stdexec::set_value(
        std::move(o_->r_), (decltype(as)&&) as...);
  }

  void set_error(auto&&) && noexcept {
    o_->_retry();
  }

  void set_stopped() && noexcept {
    stdexec::set_stopped(std::move(o_->r_));
  }
};
```

The receiver is a callback

Section 1.9.3 states: "The lack of a standard callback shape obstructs generic design" and "few of these possibilities accommodate cancellation signals."

The `_retry_receiver` above has three entry points:

- `set_value` - success callback
- `set_error` - error callback
- `set_stopped` - cancellation callback

This is a standard callback shape that accommodates cancellation signals. Standardizing this shape may be precisely the solution the prior-art section envisions.

---

5. Conclusion

A sprawling asynchronous library grounded in continuation-passing style inevitably confronts trade-offs at every level of the design - the specification serves the committee best where it confronts them too.

Acknowledgments

The author thanks Alexander Stepanov for participating in the Boost documentary and for laying the foundation for modern C++.

References

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