Coroutine-Native I/O for C++29 (The Network Endeavor)

Document Number:	P4100R1
Date:	2026-05-01
Intent:	Inform
Audience:	LEWG

Reply-to: Vinnie Falco vinnie.falco@gmail.com
Steve Gerbino steve@cppalliance.org
Michael Vandeberg michael@cppalliance.org
Mungo Gill mungo@cppalliance.org
Mohammad Nejati mohammad@cppalliance.org
C++ Alliance Proposal Team

Table of Contents

Abstract

C++ coroutines have five language mechanisms that combine into the ideal substrate for coroutine-native I/O.

Two libraries - Capy and Corosio - use those mechanisms directly to deliver type-erased streams, separate compilation, and ABI stability on C++20 today. Three independent Boost libraries are building on them. A multi-paper series, split into two stages, traces the path from published implementation to standard vocabulary. Stage One is a set of pure C++20 abstractions with no platform dependency. Stage Two is platform I/O. How those stages compose, what each paper delivers independently, and where the boundaries of a coroutine-native design lie are the subjects of this paper.

Revision History

R1: May 2026 (pre-Brno mailing)

- The buffer-concepts paper (Paper 4) is split into a buffer-ranges pair and a dynamic-buffer pair, following the IoAwaitable pattern (P4003R3 ask + P4172R0 design). Each topic now has an ask paper and a design paper.
- Dynamic Buffer is promoted to its own series slot as Paper 5. Stream Concepts becomes Paper 6 (was 5). Combinators becomes Paper 7 (was 6). Stage Two papers (Timers, Signals, Files, TCP, DNS, UDP, TLS) shift from Papers 7-13 to Papers 8-14. The series is now fourteen papers (was thirteen).
- Section 8.4 narrowed to the buffer-ranges vocabulary only; new Section 8.5 added for Dynamic Buffer. Subsequent Section 8.x headings renumbered.
- Section 11 timeline tables updated with Dynamic Buffer entries and shifted Stage Two paper numbers.
- Added Section 2.3 (Early Performance Evidence) with preliminary Boost.Redis benchmark results comparing Corosio, Asio, Rust, and Go clients.

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 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.

The author has published Boost libraries and has a stake in the project's success.

This paper asks for nothing.

---

2. The Evidence

Technical and performance claims in this document are backed by published code. Every paper in the series is accompanied by a published library that implements it, a test suite, benchmarks where performance claims are made, and cross-platform CI. The committee can clone the repo and run the tests and examples.

2.1 What We Built

Two libraries represent the work to be standardized. Neither requires Boost:

Library	Role	Status
Capy	Abstract layer: execution model, buffer concepts, stream concepts, concurrency primitives	Published. Pure C++20. No platform dependency
Corosio	Platform layer: sockets, timers, DNS, TLS, signals	Published. IOCP, epoll, kqueue, select. Requires Capy

Software building on this foundation:

Library	Role	Status
Boost.Http	HTTP/1.1 server, sans-I/O	In development. Compiled once against `any_stream`, ABI-stable
Boost.Burl	HTTP client	In development
Boost.Beast2	Successor to Boost.Beast	In development
Boost.WebSocket	WebSocket protocol, sans-I/O	Planned

2.2 Independent Adopters

These libraries are maintained by other Boost authors:

Library	Author	Status
Boost.MySQL	Ruben Perez	v2 planned on Capy/Corosio; Redis proof-of-concept validates the path
Boost.Redis	Marcelo Zimbres Silva	Experimental port completed; conversion reports published on the Boost Developers Mailing List. Early benchmark results in Section 2.3
Boost.Postgres	(upcoming)	Building on Corosio from day one

A production trading infrastructure company is evaluating Corosio for high-performance networking. This is money on the wire. P4125R0 [1] documents qualitative findings from a derivatives exchange porting from Asio callbacks to coroutine-native I/O.

2.3 Early Performance Evidence

Marcelo Zimbres Silva (Boost.Redis maintainer) benchmarked Ruben Perez's Corosio port of Boost.Redis against the Asio-based original and two non-C++ clients. The metric is wall-clock time multiplied by client CPU utilisation, normalised to the Corosio result:

Client	Normalised time
`boost_redis_corosio`	1.00
`boost_redis_asio_cb`	1.35
`boost_redis_asio_co`	1.73
`redis_rs` (Rust)	4.62
`go_redis` (Go)	22.33

No profiling has been performed and the source of the improvement is not yet characterised. The benchmark suite is published at redis-cli-comp [2]; further profiling and characterisation are planned.

---

3. What We Found

Asio got many things right. We built on its stream model, its buffer sequences, its executor architecture. We started from C++20.

C++20 coroutines were designed for generality: async patterns, lazy evaluation, generators. We used them directly for I/O. Five properties combine into a substrate suited to I/O. P4088R0 [3] traces the full causal chain from mechanism to library:

1. **Type erasure is structural.** `coroutine_handle<>` erases the coroutine's type. The frame is the erasure. `task<T>` has one template parameter. `any_stream` works without per-operation allocation. The compiler provides the type erasure that template-based designs must build by hand.
2. **Promise customization.** `promise_type` controls allocation, suspension, resumption, and error handling - exactly the control points I/O needs. The IoAwaitable protocol uses `await_suspend` to propagate the executor, stop token, and frame allocator. No language extensions required.
3. **Stackless frames.** Each coroutine frame is independently allocated, independently suspendable, independently resumable. This maps directly to how operating systems do I/O: each operation suspends independently, the kernel completes them in any order, the reactor resumes them individually.
4. **Symmetric transfer.** `await_suspend` returning `coroutine_handle<>` enables zero-overhead context switching. If already on the correct thread, return the handle for direct resumption; otherwise post and return `noop_coroutine()`. No stack buildup.

5. **Frame as state.** Local variables survive across suspension in the compiler-generated frame. The coroutine frame is always allocated. The state it carries is free relative to the frame cost. This subsidizes type erasure: `any_stream` wraps any transport behind a vtable, and the operation state lives in the caller's frame, which already exists.

The five properties converge on problems that are specific to C++:

Problem	Why C++-specific	Coroutine resolution
Template explosion	Go has no templates	`coroutine_handle<>` type erasure eliminates template metaprogramming
Compile times	Rust does not have this problem at C++ scale	`any_stream` compiles once and ships as a binary
Allocation control	Python has no user-facing allocator model	Frame allocator propagation gives users control where it matters
ABI stability	The committee's twenty-year wound	`any_stream`'s ABI is a vtable - the same contract since 1983

Five independent language mechanisms. One I/O design where correctness, ergonomics, and performance do not trade against each other.

---

4. Design Criteria

Eight requirements for I/O in the standard, and what the design provides for each.

4.1 Zero overhead for features not used

The three-layer architecture (Section 6) gives every I/O object an abstract layer (virtual dispatch, ABI-stable), a concrete layer (protocol-specific, separately compilable), and a native layer (templated, fully inlined). The user who needs zero overhead uses the native layer. The user who needs ABI stability uses the abstract layer. Neither pays for the other.

4.2 Tools for building, not built-in solutions

Stream concepts are tools. Buffer concepts are vocabulary. The series proposes the abstractions that HTTP and WebSocket libraries are built from - not the HTTP or WebSocket libraries themselves. Boost.Http, Boost.MySQL, and Boost.Redis are building on these abstractions today.

4.3 ABI stability

`any_stream` type-erases a fixed set of operations. Those operations have not changed in forty years. Libraries that accept `any_stream&` compile once and ship as `.so` / `.dll` / `.a` files. New transports plug in without recompilation.

When a coroutine calls `co_await stream.read_some(buf)`, the caller's frame persists across suspension. That frame is already allocated. `any_stream` type-erases without per-operation allocation because the operation state lives in the coroutine frame, not in a template. The frame allocation we cannot avoid subsidizes the type erasure we want.

4.4 Backward compatibility

Pure C++20. No language extensions. No compiler intrinsics. Works with every conforming compiler today.

4.5 Production deployment

Cross-platform: IOCP, epoll, kqueue, select. Three independent Boost library adopters (MySQL, Redis, Postgres). One institutional evaluation in production trading infrastructure (P4125R0 [1]).

4.6 Modularity

Each paper proposes the narrowest abstraction that delivers value on its own. Stage One papers have no platform dependency. Paper 4 (buffer ranges) and Paper 5 (dynamic buffer) have no async dependency. Each paper depends only on what came before. No paper requires unfinished work by a different author.

4.7 Language-library co-design

The IoAwaitable protocol exploits `promise_type`, `await_suspend`, and symmetric transfer to propagate execution context, stop tokens, and frame allocators. C++20 coroutines were designed as language features. This series uses them as library infrastructure.

4.8 Convergence with existing practice

Ecosystem	Read	Write	Buffer	Scatter/gather
BSD (1983)	`read()`	`write()`	`void*` + len	`readv`/`writev`
POSIX	`readv()`	`writev()`	`iovec`	`iovec[]`
Asio	`async_read_some()`	`async_write_some()`	`const_buffer`	`ConstBufferSequence`
Go	`io.Reader`	`io.Writer`	`[]byte`	`io.ReadFrom`
Rust	`AsyncRead`	`AsyncWrite`	`&[u8]`	`vectored`
.NET	`Stream.ReadAsync()`	`Stream.WriteAsync()`	`Memory<byte>`	`ReadOnlySequence<byte>`
Ours	`ReadStream::read_some()`	`WriteStream::write_some()`	`const_buffer`	`ConstBufferSequence`

Seven independent ecosystems. Same operations. Same shape.

---

5. Asio Continuity

This work builds on Asio, and we acknowledge that debt.

Asio has been used in production worldwide for over twenty years. It is the foundation of the Networking TS. Boost.Beast, Boost.MySQL, Boost.Redis, and hundreds of proprietary codebases depend on it. The operations it models - `async_read_some`, `async_write_some`, buffer sequences, executors - are the same operations every I/O framework arrives at independently.

The shift from C++11 to C++20 changes what is possible. Asio's named requirements become C++20 concepts. Its callback-based completion tokens become coroutine awaitables. Its stream model - the same `read_some` / `write_some` pair - gains structural type erasure through coroutine frames. The operations have not changed. The language has.

5.1 The Asio Adapter

Papers 1 and 2 deliver immediate value to existing Asio users. A small adapter wraps any Asio executor to satisfy the `Executor` concept. Asio users get `task<T>` as a drop-in replacement for `asio::awaitable<T>`.

Three gains:

1. **Physical insulation.** `asio::awaitable<T>` forces the Asio dependency into every header that names the return type. `task<T>` confines the dependency to the `.cpp` file that touches the socket. The header declares `task<response> handle_request(any_stream&)`. Consumers include neither Asio nor Corosio.

2. **Compilation improvement.** Fewer translation units include Asio headers. The heavy machinery - executor model, completion tokens, socket options - is confined to the files that need it.
3. **Backend insulation.** Swap Asio for Corosio by recompiling one `.cpp` file. Consumers never recompile. Headers do not change.

Frame allocator propagation works through the adapter. Recycling allocators on MSVC yield a 3.1x throughput improvement (P4007R3 [4] Section 5).

5.2 The Migration Path

Not "rewrite your application." Recompile one file.

The business logic accepts `any_stream&` and returns `task<T>`. The `.cpp` file that creates the socket includes Asio (or Corosio, or any conforming backend). The compilation boundary is the migration boundary. Cross it at your own pace.

---

6. The Three-Layer Architecture

Every I/O object exposes three API layers.

Abstract. `io_stream`, `io_read_stream`, `io_write_stream`. Protocol-agnostic, virtual dispatch, separately compilable, no platform headers.

Concrete. `tcp_socket`, `timer`, `signal_set`. Full protocol-specific API, still virtual dispatch, still separately compilable.

Native. `native_tcp_socket<Backend>`. Templated on the platform backend. Member function shadowing eliminates the vtable. Full inlining.

The three layers share a single inheritance chain:

```
         io_stream                  abstract (Layer 3)
             |
        tcp_socket                  concrete (Layer 2)
             |
native_tcp_socket<Backend>          native   (Layer 1)
```

Property	Abstract	Concrete	Native
Compilation speed	Fastest	Fast	Slowest (platform headers + templates)
Separate compilation	Yes	Yes	No
Call overhead	Virtual dispatch	Virtual dispatch	None (direct / inlined)
API surface	Protocol-agnostic (bytes)	Full protocol-specific API	Same as concrete, fully inlined
Use case	Libraries, generic algorithms	Application code	Hot paths, benchmarks

The user chooses the layer. Neither forces a choice on the other.

---

7. Approach to Standardization

We are:

- Users first,
- Implementors second, and
- Proposers third.

The series is layered: each paper builds on the last, each is backed by implementation experience, and each delivers value on its own. Two stages mirror the library split. The Stage One papers correspond to Capy; the remainder correspond to Corosio.

7.1 Stage One: Pure C++ Abstractions

#	Paper	Abstraction
1	**IoAwaitable Protocol**	Coroutine execution protocol, executor model
2	**Coroutine Task**	`task<T>`, launch functions, `thread_pool`, `system_context`
3	**Executor Utilities**	`strand`, `any_executor`
4	**I/O Buffer Ranges**	Byte-region descriptors and range-based scatter/gather concepts
5	**Dynamic Buffer**	Growable byte buffer with prepare/commit/data/consume
6	**Stream Concepts**	Coroutine stream concepts for async byte I/O
7	**Combinators**	`when_all`, `when_any`

Pure C++20. No platform code. These abstractions enable sans-I/O protocols in the ecosystem: HTTP, WebSocket, TLS wrappers.

Stage One alone delivers the vocabulary for the entire async I/O ecosystem. External libraries implement portable, platform-specific I/O in terms of `std::io`. The ecosystem delivers the platform. The standard delivers the vocabulary.

7.2 Stage Two: Platform I/O

#	Paper	Abstraction
8	**Timers**	Async timer operations
9	**Signals**	Async signal handling
10	**Files**	Async file I/O
11	**TCP**	Sockets, acceptors, endpoints, IP addresses
12	**DNS**	Async name resolution
13	**UDP**	Datagram sockets
14	**TLS**	Transport security wrappers

The first three papers in Stage Two have nothing to do with networking. Many papers can proceed before the word "socket" appears.

7.3 Risk Separation

Previous approaches focused on getting sockets right before anything ships. This approach gets the abstractions right and lets sockets follow.

Stage One defines the concepts. Stage Two provides implementations. Between them, the ecosystem experiments. Multiple socket implementations can coexist: Corosio's, an io_uring-native one, an IOCP-optimized one, a minimal embedded one. They all satisfy the same `Stream` concepts.

The ABI stability of Stage One provides the boundary. `any_stream` is the contract. The socket behind it can be replaced. Business logic never recompiles.

If the committee pursues this path, the concepts should ship first. Let the ecosystem discover the best implementations. Standardize what works.

---

8. The Paper Series

8.1 Paper 1: IoAwaitable Protocol

P4003R3 [5] published. P4172R0 [6] provides design rationale. Targeting first LEWG review at Brno (June 2026).

The coroutine execution model for I/O. Depends on nothing.

Key types. `IoAwaitable` concept, `IoRunnable` concept, `Executor` concept (`dispatch`, `post`, `context`), `execution_context` with service registry and frame allocator ownership, `executor_ref` (two-pointer type-erased executor), `io_env` (bundles executor, stop token, frame allocator).

What coroutines provide. The IoAwaitable protocol solves frame allocator timing through forward propagation via TLS. The frame allocator is available before `operator new` executes. No language extensions required. 3.1x throughput improvement using recycling allocators on MSVC (P4007R3 [4] Section 5).

Shipping status. Capy implements the full protocol. Corosio builds a complete networking stack on it. Shipping today on Windows, Linux, and macOS.

Standalone value. The protocol layer that every subsequent paper builds on. The Asio adapter (Section 5.1) wraps any Asio executor to satisfy the `Executor` concept. The IoAwaitable protocol is the interoperability point: any coroutine library that satisfies it composes with the rest of the series.

8.2 Paper 2: Coroutine Task

The concrete coroutine task type, launch functions, and execution contexts. Depends on Paper 1.

Key types. `task<T>` (lazy coroutine task, one template parameter), `run()` (blocks until the task completes), `run_async()` (submits the task for execution), `thread_pool` (multi-threaded execution context), `system_context` (process-wide singleton

backed by a `thread_pool`).

What coroutines provide. `task<T>` has one template parameter because `coroutine_handle<>` erases the coroutine's type structurally. No allocator parameter, no executor parameter, no scheduler parameter. The IoAwaitable protocol propagates context through `await_suspend` - the task type stays simple. `run()` and `run_async()` bridge synchronous and asynchronous code at the top level.

Shipping status. Shipping in Capy. Every Corosio example and test uses `task<T>` with `run()`.

Standalone value. After Papers 1 and 2, Asio users get `task<T>` with frame allocator propagation as a drop-in replacement for `asio::awaitable<T>`. Business logic drops the Asio header dependency. The I/O backend becomes swappable behind a compilation boundary (Section 5.1).

8.3 Paper 3: Executor Utilities

Serialization and type erasure over any executor. Depends on Paper 1.

Key types. `strand<Ex>` (serializes coroutine execution over any executor), `any_executor` (owning type-erased executor).

What coroutines provide. `strand` serializes access to shared state without locks. In a coroutine-native model, strands serialize coroutine resumption rather than completion handler dispatch. `any_executor` type-erases any executor behind an owning handle, enabling polymorphic executor storage.

Shipping status. Shipping in Capy.

Standalone value. `strand` and `any_executor` are general-purpose utilities that work over any `Executor`. They have no I/O dependency and no platform dependency. Any executor-based system benefits from serialization and type erasure.

8.4 Paper 4: I/O Buffer Ranges

I/O Buffer Ranges and *I/O Buffer Ranges: Design Rationale* (companion ask + design pair, document numbers TBD). Buffer descriptor and sequence vocabulary for scatter/gather I/O. Depends on nothing. No async dependency. No coroutines. No executor. Pure vocabulary.

Key types. `const_buffer` and `mutable_buffer` (byte-region vocabulary), `ConstBufferSequence` and `MutableBufferSequence` (range concepts for scatter/gather), `buffer_size`, `buffer_empty`, `buffer_length`, `buffer_copy` (customization point objects).

The growable buffer (prepare/commit) concept that R0 of this paper bundled with the buffer ranges has been split into Paper 5; see Section 8.5.

Relationship to `std::ranges`. `ConstBufferSequence` is defined as `std::ranges::bidirectional_range<T>` with a value type constraint. A single `const_buffer` is not a `std::ranges::range` - it represents a byte region, not an iterable sequence. `ranges::size` counts elements; `buffer_size` sums bytes. `ranges::drop` operates at element granularity; buffer slicing operates at byte granularity across element boundaries. The extension beyond `std::ranges` is minimal and specific.

Convergence. POSIX `iovec`, Windows `WSABUF`, Asio `const_buffer`, libuv `uv_buf_t`, Go `[]byte`, .NET `Memory<byte>`. Six ecosystems, same shape, none of them `span`.

Shipping status. Shipping in Capy. Used by every Corosio I/O operation.

Standalone value. Today, every C++ project that does I/O invents its own buffer types. Standard buffer concepts create shared vocabulary: a database driver that accepts `MutableBufferSequence` works with any I/O stack that speaks the same concepts. This paper advances independently of Paper 1 on a parallel track.

8.5 Paper 5: Dynamic Buffer

Dynamic Buffer and *Dynamic Buffer: Design Rationale* (companion ask + design pair, document numbers TBD). Growable byte buffer with two-phase write and two-phase read. Depends on Paper 4. No async dependency.

Key concept. `DynamicBuffer` (member typedefs `const_buffers_type` and `mutable_buffers_type` satisfying the buffer-ranges concepts; member functions `prepare(n)`, `commit(n)`, `data()`, `consume(n)`, `size()`, `max_size()`, `capacity()`).

Two-phase model. `prepare(n)` returns at least `n` bytes of writable space as a `mutable_buffers_type`. The caller fills bytes. `commit(m)` makes the first `m` written bytes readable through `data()` (which returns a `const_buffers_type`). `consume(k)` discards `k` readable bytes from the front. The same object holds both the writable and readable boundaries.

Convergence. Asio's `DynamicBuffer` named requirement (2003), the Networking TS named requirement (2018), .NET `IBufferWriter<T>` (2018), Go `bytes.Buffer` (2012). Three ecosystems with a two-phase model; the shapes are the same.

Shipping status. Shipping in Capy. Four implementations: `flat_dynamic_buffer`, `circular_dynamic_buffer`, `vector_dynamic_buffer`, `string_dynamic_buffer`. The first two are caller-owned-storage; the latter two adapt existing standard containers.

Standalone value. Protocol parsers, HTTP message readers, TLS record buffers, and streaming decoders all need a growable buffer with FIFO semantics. `std::vector<std::byte>` provides one half of that; the dynamic buffer concept is the shape that admits flat, ring, and adapter implementations under one interface.

8.6 Paper 6: Stream Concepts

Coroutine stream concepts for async byte I/O. Depends on Papers 1, 4, and 5.

This paper gives C++ ABI-stable I/O.

Two families of concepts. Caller-owned buffers: `ReadStream` (`read_some`), `WriteStream` (`write_some`), `Stream` (both), refined into `ReadSource` (complete reads) and `WriteSink` (complete writes, `write_eof()`). Callee-owned buffers: `BufferSource` (`pull`/`consume`) and `BufferSink` (`prepare`/`commit`). Zero-copy when needed. Simple path when not.

Type-erasing wrappers. `any_stream`, `any_read_stream`, `any_write_stream`, `any_read_source`, `any_write_sink`, `any_buffer_source`, `any_buffer_sink`. One pointer, one vtable, zero per-operation allocation. Boost.Http uses `any_stream` throughout - compiled once, linked against, works with any transport.

Convergence. BSD `read`/`write`, POSIX `readv`/`writev`, Asio `async_read_some`/`async_write_some`, Go `io.Reader`/`io.Writer`, Rust `AsyncRead`/`AsyncWrite`, .NET `Stream.ReadAsync`/`WriteAsync`. Every language arrived at the same pair of operations.

Synchronous streams for free. `co_await` checks `await_ready()` first. If `true`, no suspension. Memory buffers, test mocks, zlib decompressors, base64 decoders all satisfy `ReadStream` by returning immediately-ready awaitables. A pipeline of `tcp_socket` to `tls_stream` to `decompression_stream` to HTTP parser works regardless of which layers suspend. The algorithm does not know the difference.

Shipping status. Shipping in Capy. Corosio's `tcp_socket`, `tls_stream`, and `io_stream` all satisfy `Stream`.

Standalone value. Paper 6 completes the bridge to existing I/O ecosystems. After Papers 1-6, an `asio::ip::tcp::socket` becomes an `any_stream` through one adapter in one `.cpp` file. Business logic compiles against the standard. The Asio socket is behind the wrapper. The bulk of a networking application's code compiles against `std::io`.

8.7 Paper 7: Combinators

Structured concurrency primitives for coroutine-native code. Depends on Paper 2.

Key types. `when_all(awaitables...)` (awaits all; returns when every awaitable completes), `when_any(awaitables...)` (awaits any; returns when the first awaitable completes and cancels the rest).

Shipping status. Shipping in Capy. `when_all` and `when_any` used in Corosio's `tcp_server` for concurrent accept loops.

Standalone value. Paper 7 completes Stage One. After Papers 1-7, a user can write a full concurrent networking application against the standard: `task<>` coroutines accepting `any_stream&`, standard `DynamicBuffer` for parsing, `thread_pool` for execution, `strand` for serialization, `when_all`/`when_any` for concurrency. The standard provides the vocabulary; adapters bridge to the I/O backend. When Stage Two ships, the adapters become unnecessary.

8.8 Papers 8-10: Timers, Signals, Files

Paper 8: Timers. `timer` with `wait()`, `expires_at()`, `expires_after()`, `cancel()`, `cancel_one()`. Uses `std::chrono::steady_clock`. The simplest kernel interaction that proves the IoAwaitable protocol works end-to-end with a real operating system. Shipping in Corosio. Cross-platform.

Paper 9: Signals. `signal_set` with `add()`, `remove()`, `clear()`, `wait()`, `cancel()`. POSIX signal flags (`SA_RESTART`, `SA_NOCLDSTOP`, etc.) are supported via a `flags_t` parameter. Signals complete the server lifecycle: `co_await when_all(accept_loop(), signal_set.wait())` is graceful shutdown in one line. `<csignal>` is from 1989 and nearly unusable with modern C++. Shipping in Corosio.

Paper 10: Files. Async file I/O with `read_some()`, `write_some()`, `open()`, `close()`, and positioning. Two concrete types: `random_access_file` (seekable) and `stream_file` (sequential). Both satisfy `Stream`. The same generic algorithms work on files and sockets. On Windows, IOCP with `FILE_FLAG_OVERLAPPED`. On Linux, `io_uring` is on the roadmap; POSIX file I/O is the current backend. Shipping in Corosio. Cross-platform.

8.9 Papers 11-13: TCP, DNS, UDP

Paper 11: TCP. `tcp_socket`, `tcp_acceptor`, `endpoint`, `ipv4_address`, `ipv6_address`. A `tcp_socket` satisfies `Stream`. This delivers what the committee has been trying to standardize since N1925 [7] (2005). Shipping in Corosio. Used by other Boost libraries and users.

Paper 12: DNS. `resolver` with `resolve(host, service)` for forward resolution and `resolve(endpoint)` for reverse resolution. Without DNS, TCP sockets can only connect to hardcoded IP addresses. Shipping in Corosio.

Paper 13: UDP. `udp_socket` with `send_to()`, `receive_from()`, `bind()`, and multicast support. UDP is the transport for DNS, QUIC/HTTP/3, game networking, media streaming, and IoT protocols. Shipping in Corosio. Cross-platform.

Unix domain sockets. `local_stream_socket`, `local_stream_acceptor`, `local_datagram_socket`. Unix domain sockets provide efficient inter-process communication without the network stack overhead. Shipping in Corosio on POSIX platforms.

8.10 Paper 14: TLS

Transport security wrappers: `tls_context` for portable certificate management and `tls_stream` for encrypted I/O. A `tls_stream` wraps any `Stream` and satisfies `Stream` itself. The cryptographic engine is implementation-defined. ABI-stable by design.

The standard specifies the shape. The platform provides the encryption.

What the standard does. Portable certificate management (`tls_context`): certificates, trust anchors, verification modes, access to OS certificate store and revocation infrastructure. ABI-stable encrypted I/O (`tls_stream`): `handshake`, `read_some`, `write_some`, `shutdown`.

What the standard does not do. It does not reimplement OpenSSL. It does not mandate a specific engine. The cryptographic engine is implementation-defined - same as `std::filesystem`'s underlying OS calls. A TLS vulnerability disclosed on Monday is patchable by Tuesday. The fix does not wait for a standard library release cycle.

Convergence. Go `crypto/tls`, Rust `rustls`/`native-tls`, Python `ssl`, .NET `SslStream`, Java `SSLSocket`. Every language wraps TLS the same way: configuration object plus encrypted stream.

Shipping status. Shipping in Corosio with OpenSSL and WolfSSL backends. Both engines use the same abstract `tls_context` API.

The risk profile is lopsided: the implementation carries the risk; the interface carries none. The wrapper API has been stable since SSLv3 in 1996. Standardize the riskless part. Offload the risky implementation to the OS and ecosystem.

---

9. `std::execution`

`std::execution` is an achievement. It serves its domains well.

9.1 What It Provides

Sender algorithms compose at compile time with full type information. The optimizer sees everything. Scheduler-based dispatch unifies CPU, GPU, and I/O scheduling under one vocabulary. The committee invested years in this work. It delivers real value for compile-time work graphs and heterogeneous dispatch.

9.2 Domain Separation

The question is whether byte-oriented I/O benefits from the same model.

The properties in Section 3 - type erasure, frame allocation, symmetric transfer - work because coroutines are used directly. A sender layer between the coroutine and the platform loses them. P4007R3 [4] documents four structural gaps where `std::execution` meets coroutines: three at the boundary - error reporting, error returns, and frame allocator propagation - and one inside the composition mechanism: the symmetric transfer gap (documented in P2583R4 [8]).

Domain specialization is not fragmentation. GPU compute got `nvexec` with CUDA extensions and a separate namespace. C++ has multiple container types, multiple string types, multiple smart pointer types. Two async models for two distinct domains is the same principle.

9.3 Convergence Points

The stream concepts impose no async-model dependency. A sender-based I/O library could define a `sender_stream` that satisfies `ReadStream` / `WriteStream` by returning sender-based awaitables. The concepts provide byte-oriented stream vocabulary that the entire ecosystem - including the sender ecosystem - can adopt.

The buffer concepts (Papers 4 and 5) have no async dependency at all. They are pure vocabulary for every I/O proposal.

9.4 Diversification Protects the Timeline

If the committee relies exclusively on one async path for networking and that path encounters delays, the networking timeline slips again. This has happened before.

Stage One carries minimal technical risk if allowed to proceed in parallel. Pure C++20 with no platform dependency. If sender-based networking succeeds, the buffer concepts and stream concepts enrich it. Nothing is wasted. If it encounters delays, the committee has an independent path ready.

Letting both paths proceed is the conservative choice.

---

10. Limitations

A coroutine-native I/O design has boundaries. Stating them plainly is part of the report.

- **Compile-time work graphs.** Coroutine-native I/O is a runtime model. It does not express static dataflow graphs where the compiler can reason about the entire pipeline at compile time. `std::execution` serves this domain.

- **Heterogeneous dispatch.** GPU compute, NUMA-aware scheduling, and cross-device dispatch require the scheduler abstraction that `std::execution` provides. Coroutine executors handle thread-pool and strand-based dispatch. They do not target hardware heterogeneity.
- **Cooperative scheduling.** The model assumes cooperative multitasking. Preemptive multitasking is outside scope.
- **C library dependencies.** TLS wrappers depend on C libraries (OpenSSL, WolfSSL). The standard does not eliminate that dependency. It provides the stream abstraction that makes such wrappers composable and replaceable.

These boundaries define where coroutine-native I/O ends and where other models provide value.

---

11. Timeline

Target: all papers through LEWG by end of 2028. LWG wording through 2029H1.

11.1 Phase 1: Stage One Foundation (2026)

Quarter	Paper	Milestone
Q1 2026	IoAwaitable Protocol	P4003R1 published
Q2 2026	IoAwaitable Protocol	First LEWG review at Brno
Q3 2026	Coroutine Task	Paper published
Q3 2026	Executor Utilities	Paper published
Q3 2026	I/O Buffer Ranges	Paper published
Q3 2026	Dynamic Buffer	Paper published
Q4 2026	Coroutine Task	LEWG review
Q4 2026	Stream Concepts	Paper published
Q4 2026	Combinators	Paper published
Q4 2026	Executor Utilities	LEWG review
Q4 2026	I/O Buffer Ranges	LEWG review
Q4 2026	Dynamic Buffer	LEWG review

11.2 Phase 2: Stage One Completion + Stage Two Begins (2027 H1)

Quarter	Paper	Milestone
Q1 2027	Stream Concepts	LEWG review
Q1 2027	Combinators	LEWG review
Q1 2027	Timers	Paper published
Q2 2027	Signals, Files	Papers published
Q2 2027	Timers	LEWG review

11.3 Phase 3: Networking (2027 H2)

Quarter	Paper	Milestone
Q3 2027	TCP, DNS	Papers published
Q3 2027	Signals, Files	LEWG review
Q4 2027	UDP	Paper published
Q4 2027	TCP	LEWG review

11.4 Phase 4: Completion (2028)

Quarter	Paper	Milestone
Q1 2028	DNS, UDP	LEWG review
Q2 2028	TLS	Paper published
Q3 2028	TLS	LEWG review
Q4 2028	Full series	LWG wording review begins

---

12. What We Continue to Maintain

We continue to maintain Capy and Corosio after standardization.

Production C++ lags the standard by three to seven years. A feature that lands in C++29 is not available to most production users until 2032-2036. Users who adopt Capy and Corosio today on C++20 need the Boost versions for years after the standard ships. This is the Boost.Filesystem pattern: precede the standard, coexist with it, persist for users on older compilers. Migration happens at the user's pace.

---

13. `std::io`

```
std::io::task<>
handle_http_request( std::io::any_stream& stream )
{
    auto [ec, req] = co_await boost::http::read_request( stream );
    if( ! ec )
    {
        auto res = process_request( req );
        std::tie(ec) = co_await boost::http::write_response( stream, res );
    }
    co_return { ec };
}
```

We built this. It works. We are reporting what we found.

References

[1] P4125R0 - "Field Report: Coroutine-Native I/O at a Derivatives Exchange" (Vinnie Falco, 2026).

[2] redis-cli-comp (Marcelo Zimbres Silva, 2026).

[3] P4088R0 - "What C++20 Coroutines Already Buy The Standard" (Vinnie Falco, 2026).

[4] P4007R3 - "Senders and Coroutines" (Vinnie Falco, Mungo Gill, 2026).

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

[6] P4172R0 - "IoAwaitable for Coroutine-Native Byte-Oriented I/O" (Vinnie Falco, Steve Gerbino, Mungo Gill, 2026).

[7] N1925 - "Networking proposal for TR2 (rev. 1)" (Gerhard Wesp, 2005).

[8] P2583R4 - "Symmetric Transfer and Sender Composition" (Mungo Gill, Vinnie Falco, 2026).

Acknowledgments

Chris Kohlhoff designed Asio's stream model, buffer sequences, and executor architecture. Twenty years of production deployment is the foundation this work builds on.

The authors of `std::execution` - Eric Niebler, Kirk Shoop, Lewis Baker, and their collaborators - invested years in a principled async framework for C++. Their work on sender/receiver serves domains where compile-time composition and heterogeneous dispatch matter. We respect that achievement and build alongside it.

The committee designed C++20 coroutines. Gor Nishanov, Lewis Baker, and their collaborators gave C++ the language mechanisms that make this I/O design possible.

Dietmar Kühl's feedback on P4007R3 and P4014R2 improved the technical quality of both papers.

Ruben Perez, Marcelo Zimbres Silva, and the Boost.Postgres team adopted Capy and Corosio independently. Their experience is evidence we could not manufacture ourselves.