Document	**D4008R0**
Date:	2026-02-09
Reply-to:	Vinnie Falco <vinnie.falco@gmail.com>
Audience:	All of WG21

The C++ Standard Cannot Connect to the Internet

Abstract

This paper discusses a key observation: The C++ Standard cannot connect to the Internet. The problem is not missing sockets. Third-party networking libraries exist. The problem is that C++ lacks a standard asynchronous execution model designed for I/O, which prevents asynchronous algorithms from composing across library boundaries. Every other major programming language solved this problem by standardizing an async I/O foundation, and the result was an explosion of higher-level frameworks. C++ has no such ecosystem because there is no agreed-upon foundation to build on. This paper examines the evidence, proposes three tests that a standard async model should meet, and asks the committee to consider whether networking deserves higher priority than it currently receives.

1. The Observation

The C++ standard cannot connect to the internet, and it is not for lack of trying. Third-party networking libraries exist and are widely used. Boost.Asio (https://www.boost.org/doc/libs/release/doc/html/boost_asio.html) has been in production for over twenty years. But Asio itself ships as two incompatible published versions, Boost.Asio and standalone Asio (https://github.com/chriskohlhoff/asio), with different namespaces and different build configurations. This is the coordination problem in miniature: even the most popular, most mature C++ networking library cannot agree with itself on a single interface.

The deeper problem is not sockets. It is the absence of a standard asynchronous execution model for I/O.

If a programmer wants to write a composable async algorithm, for example an HTTP conversation, a WebSocket message exchange, or an asynchronous file transfer with optional compression, what execution model should they use? There is no answer. There are Asio completion tokens, Meta's folly::coro (https://github.com/facebook/folly/blob/main/folly/experimental/coro/README.md), Qt signals and slots (https://doc.qt.io/qt-6/signalsandslots.html), custom callback systems, promise/future chains, and ad-hoc event loops. Each is an island. Algorithms expressed in one model cannot compose with algorithms from another.

Not every committee member works with network programming daily. WG21 is full of experts in language design, template metaprogramming, numerics, concurrency, safety, and many other domains. This paper is written from a networking practitioner's perspective, for colleagues who may not have the same daily experience with this problem. It aims to explain, with evidence, why the networking gap matters and what it costs.

2. The Tower of Abstraction

Every major programming language standardized an async foundation. In some cases this was a full I/O model in the standard library. In others it was a narrow composability trait that third-party runtimes build on. Either way, the standardized foundation enabled tall towers of abstraction. The foundation came first. The frameworks followed.

Python standardized `asyncio` in Python 3.4 (2014) via PEP 3156 (https://peps.python.org/pep-3156/), providing both the async model and I/O integration in the standard library. The ecosystem built upward. Django (https://github.com/django/django) (86.7k GitHub stars), the web framework "for perfectionists with deadlines," sits atop this foundation.

JavaScript standardized `Promise` in ES2015 (June 2015) and `async` / `await` in ES2017 (TC39 proposal (https://tc39.es/proposal-async-await/)). The language standard defines the composability primitive; the I/O runtime comes from the host environment (Node.js, browsers, Deno). Because every runtime implements the same `Promise` contract, libraries like Express.js (https://github.com/expressjs/express) (68.7k stars) and Next.js (https://github.com/vercel/next.js) (137.6k stars) compose across all of them.

Go shipped goroutines, channels, and `net/http` in the standard library from day one (2009). The `net/http` package is imported by 1,705,800 known packages (https://pkg.go.dev/net/http). The entire Go microservices ecosystem is built on this foundation.

Rust stabilized `async` / `await` in Rust 1.39 (November 2019; Rust blog announcement (https://blog.rust-lang.org/2019/11/07/Async-await-stable/)) and standardized the `Future` (https://doc.rust-lang.org/std/future/trait.Future.html) trait in `std`. The I/O runtime is not in the standard library; Tokio (https://github.com/tokio-rs/tokio) (31k stars) fills that role. But because every runtime implements the same `Future` trait, libraries like hyper (https://github.com/hyperium/hyper) and Axum (https://github.com/tokio-rs/axum) (24.9k stars) are runtime-agnostic. Rust standardized the narrow composability contract, and the ecosystem built on it.

Java has had `java.net` in the standard library since JDK 1.0 (1996). The ecosystem built upward. Spring Boot (https://github.com/spring-projects/spring-boot) (79.9k stars) sits atop this foundation.

C# shipped `async` / `await` and the `Task` type in C# 5.0 (2012; overview (https://dotnetcurry.com/csharp/869/async-await-csharp-dotnet)), with a built-in thread pool and I/O completion in the standard library. The ecosystem built upward. ASP.NET Core (https://github.com/dotnet/aspnetcore) (37.7k stars) sits atop this foundation.

In every case, standardization of an async foundation, whether a full I/O model or a narrow composability trait, enabled the ecosystem to build upward. The frameworks came after the foundation. The combined GitHub stars of these higher-level frameworks exceed 500,000.

C++ added coroutines in C++20 but standardized neither an async I/O model nor a composability trait for async operations. The language machinery is there. The foundation is not. C++ bottoms out at Boost.Asio (https://github.com/boostorg/asio) (1.5k stars, third party, no standard status), or raw POSIX/Winsock. There is no Django, no Express, no Spring Boot of C++. Not because C++ programmers are less capable, but because there is no standard foundation to build on.

3. The Coordination Problem

The C++ standard exists to solve coordination problems. P2000R4 (https://open-std.org/jtc1/sc22/wg21/docs/papers/2022/p2000r4.pdf) ("Direction for ISO C++") was created specifically to address concerns about C++ "losing coherency due to proposals based on differing and sometimes mutually contradictory design philosophies." The standard provides a shared foundation so that independently developed libraries can interoperate.

The async I/O domain is the textbook case where that foundation is missing.

Without a standard async model for I/O, every library invents its own. Boost.Asio (https://www.boost.org/doc/libs/release/doc/html/boost_asio.html) uses completion tokens. Meta's folly::coro (https://github.com/faceb

ook/folly/blob/main/folly/experimental/coro/README.md) is "a developer-friendly asynchronous C++ framework based on the Coroutines TS." Qt (https://doc.qt.io/qt-6/signalsandslots.html) uses signals and slots, described as "Qt's central communication mechanism between objects, serving as an alternative to callbacks." Beyond these, there are custom callback systems, promise/future chains, and ad-hoc event loops in countless codebases.

Algorithms written for one model cannot compose with another. An HTTP library built on Asio completion tokens cannot be used by code built on folly::coro. A TLS wrapper using Qt's event loop cannot plug into Boost.Beast (https://www.boost.org/doc/libs/release/libs/beast/). Each library is an island.

This fragmentation is the direct cause of the shallow abstraction tower. Nobody builds Django-scale frameworks in C++ because the foundation is not agreed upon.

The "No Dependencies" Culture

C++ libraries routinely advertise "no external dependencies" or "header-only" as a selling point. nlohmann/json (https://github.com/nlohmann/json) (48.8k stars) advertises "no external dependencies beyond a C++11 compliant compiler." cpp-httplib (https://github.com/yhirose/cpp-httplib) (16k stars) bills itself as "a C++ header-only HTTP/HTTPS server and client library" with no dependencies. fmt (https://github.com/fmtlib/fmt) (23.2k stars) highlights "no external dependencies."

This cultural norm is a symptom of the missing foundation. In a healthy ecosystem, depending on shared infrastructure is normal. In Python, depending on `asyncio` is not a liability. In C++, depending on Boost.Asio is treated as a burden.

John Lakos's *Large-Scale C++ Software Design* (Addison-Wesley, 1996 (https://informit.com/store/large-scale-c-plus-plus-software-design-9780201633627)) established that well-structured systems with clear hierarchical dependencies are "fundamentally easier and more economical to maintain, test, and reuse." The "no dependencies" instinct inverts this principle. It treats isolation as a virtue when shared foundations would be more productive.

The absence of a standard async model makes this dysfunction worse. Since no foundation exists, every library must reinvent it or avoid async entirely. cpp-httplib, the most popular C++ HTTP library listed above, warns in its own README: "This library uses 'blocking' socket I/O. If you are looking for a library with 'non-blocking' socket I/O, this is not the one that you want." A 16k-star HTTP library cannot offer async because there is no standard async foundation to build on.

4. How We Got Here

In 2014, the C++ committee decided to "adopt existing practice" for networking, basing a proposal on Boost.Asio (https://www.boost.org/doc/libs/release/doc/html/boost_asio.html) (P3185R0 (https://wg21.link/p3185r0) documents this history). Chris Kohlhoff published a reference implementation of the Networking TS (https://github.com/chriskohlhoff/networking-ts-impl).

By 2021, the executor debate had consumed the effort. On 2021-09-28, SG1 polled whether "one grand unified model" for asynchronous execution was needed. The result was no consensus: 4 SF, 9 WF, 5 N, 5 WA, 1 SA. P2453R0 (https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p2453r0.html) documents these polls. Nobody objected to portable socket wrappers. The schism was entirely about the execution model.

The committee chose P2300 (https://wg21.link/p2300) (`std::execution`), a sender/receiver framework. It is now in the C++26 working draft.

C++26 is about to ship `std::execution` , an asynchronous execution model. The committee decided it needed a standard async framework. That instinct was correct. But the framework that landed was designed for GPU and parallel computing (section 5.1 presents the evidence). The stdexec documentation (https://nvidia.github.io/stdexec/) confirms: "Interop with networking is being explored for C++29." Networking is deferred to C++29 at the earliest.

This is not blame. The Networking TS was proposed in 2014. It is now 2026. The C++ standard still cannot connect to the internet. The domain is genuinely difficult. The instinct to find a universal model is natural. But the cost of the delay is real. The committee recognized the need for a standard async model, and that recognition was right. The question is whether the model that landed serves the most important use case.

5. Three Tests for a Standard Async Model

If C++ is going to standardize an asynchronous execution model, it should meet three criteria. This section proposes those criteria and evaluates `std::execution` against each.

5.1 It Should Put Networking First

Networking is not just another feature. It is the infrastructure through which modern institutions coordinate. Samo Burja's *Great Founder Theory* (samoburja.com/gft (https://www.samoburja.com/gft/))

describes "social technology" as a coordination mechanism that can be documented and taught. The internet is the most consequential social technology of our era. Every other major programming language standardized an async foundation for networking, whether a full I/O model or a narrow composability trait (section 2). None of them ship a sender/receiver execution framework. C++ ships neither. When a language cannot participate in building the coordination infrastructure that civilization depends on, the gap is not merely technical. It is strategic.

GPU users already have CUDA, which requires NVIDIA's non-standard compiler. The `__device__`, `__global__`, and `<<< >>>` syntax are not valid C++ (CUDA C/C++ Language Extensions (https://docs.nvidia.com/cuda/cuda-programming-guide/05-appendices/cpp-language-extensions.html)). Adding `std::execution` to the standard provides no benefit to GPU users that stdexec on vcpkg (https://vcpkg.link/ports/stdexec) does not already provide.

`std::execution`'s design priorities explicitly target GPU computing. P2300R10 (https://open-std.org/jtc1/sc22/wg21/docs/papers/2024/p2300r10.html) section 1.1 frames the motivation around "GPUs in the world's fastest supercomputer." Section 1.3.2's second end-user example is "Asynchronous inclusive scan," a GPU parallel primitive using `bulk`. The `bulk` algorithm has no networking analog.

Networking creates towers. GPU does not. The tower of abstraction argument only works if standardization actually enables higher-level libraries to proliferate. For networking, the evidence is overwhelming: sockets lead to HTTP, which leads to REST, which leads to web frameworks, which lead to full-stack applications. Django, Express, Spring Boot, Axum (500k+ combined GitHub stars) all sit atop standardized async I/O foundations. Each layer builds on the one below.

The GPU ecosystem is wide but not tall. CUDA leads to cuDNN, cuBLAS, cuFFT, OptiX, and custom kernels. Each library uses the foundation directly. There is no stacking. There is no "Django of GPU." GPU workloads are domain-specific: custom kernels for physics simulations, ML training loops, rendering pipelines. The use cases do not compose upward the way networking use cases do.

Standardizing `std::execution` for GPU would not produce the ecosystem benefit that standardizing an async I/O model for networking would. The return on investment is asymmetric: networking standardization enables exponential ecosystem growth (towers), while GPU standardization enables linear growth (more libraries at the same level).

5.2 It Should Be Built on C++20 Coroutines

C++20 gave C++ a language-level mechanism for writing async code that looks like sync code. The C++20 coroutines proposal (https://wg21.link/p0912r5) (P0912R5) was a deliberate language design choice. `co_await`, `co_return`, and `co_yield` are keywords in the language. A standard async I/O framework should build on this foundation, not work around it.

`std::execution` shipped in C++26 without a coroutine task type. P3552 (https://wg21.link/p3552) ("Add a Coroutine Task Type") is still in flight, listing 12 unsolved design objectives. The primary way users are expected to write async code was not included in the framework.

There is a deeper problem. D4007R0 (https://wg21.link/p4007) ("std::execution Needs More Time") documents that `std::execution`'s backward-flow context model provides the allocator after the coroutine frame is already allocated. Eric Niebler characterizes the issue in P3826R3 (https://wg21.link/p3826): "The receiver is not known during early customization. Therefore, early customization is irreparably broken." This is a fundamental incompatibility with coroutine-based I/O, where the allocator must be known at frame allocation time.

Alternative models are coroutines-first and do not have this problem. D4003 (https://wg21.link/p4003) (IoAwaitables) flows context forward through coroutine chains, with the allocator known at the launch site. TooManyCooks (https://github.com/tzcnt/TooManyCooks) is a C++20 coroutine runtime built around `tmc::task` as the core type. Capy (https://github.com/cppalliance/capy) is a coroutine-first execution model with forward-flowing context. Each of these designs treats coroutines as the primary user interface for async code, not as an afterthought bolted onto a different execution model.

5.3 It Should Be Narrow

Successful C++ abstractions capture one essential property. Iterators abstract over traversal: "One seldom needs to know the exact type of data on which an algorithm works since most algorithms work on many similar types" (Stepanov, "The Standard Template Library," 1994 (https://stepanovpapers.com/Stepanov-The_Standard_Template_Library-1994.pdf)). Allocators abstract over memory strategy (cppreference: Allocator (https://en.cppreference.com/w/cpp/named_req/Allocator)). RAII abstracts over resource lifetime (cppreference: RAII (https://en.cppreference.com/w/cpp/language/raii)). Each leaves everything else to the user.

`std::execution` abstracts over scheduling, context propagation, error handling, cancellation, algorithm dispatch, and hardware backend selection simultaneously. P2300R10 (https://open-std.org/jtc1/sc22/wg21/docs/papers/2024/p2300r10.html) section 1.2 lists priorities including "diversity of execution resources," "cancellation," "error propagation," "where things execute," and "manage lifetimes asynchronously." That is not one essential property. That is six.

The specification size tells the story. The C++26 `[exec]` section (eel.is/c++draft/exec (https://eel.is/c++draft/exec)) spans 16 major subsections (33.1 through 33.16) with thousands of lines of normative wording. By contrast, D4003 (https://wg21.link/p4003)'s IoAwaitables wording section is roughly 680 lines. D4003's own non-normative note states:

> *"The wording below is not primarily intended for standardization. Its purpose is to demonstrate how a networking-focused, use-case-first design produces a dramatically leaner specification footprint. Compare this compact specification against the machinery required by P2300/P3826."*

A narrow model designed for networking requires a fraction of the specification that a wide model designed for everything requires. This matters for implementers, for reviewers, and for the long-term maintainability of the standard. (For a fuller treatment of the narrow-vs-wide argument, see On Universal Models section 4.)

Ten or more companion papers are still fixing `std::execution`'s design in the 2025-2026 mailings, addressing lifetimes, exception safety, diagnostics, memory bloat, forward progress guarantees, and a missing task type. A framework with this many open design questions has not yet demonstrated the stability that standardization requires.

5.4 The Standard Might Not Even Need Sockets

A further observation: if the standard adopts an async execution model that is opinionated on buffer-oriented I/O, it might not need to standardize sockets at all. Users would choose a third-party library for the transport (Asio sockets, io_uring, platform APIs), but they could express the bulk of their algorithms abstractly against standard buffer concepts. Those algorithms would be sharable across the ecosystem regardless of the transport underneath.

This is the model already demonstrated in practice.

Boost.HTTP follows a Sans-I/O architecture. Its documentation states: "The library itself does not perform any I/O operations or asynchronous flow control. Instead, it provides interfaces for consuming and producing buffers of data and events." It works with "any I/O framework (Asio, io_uring, platform APIs)" (Boost.HTTP (https://github.com/cppalliance/http)).

Boost.Capy provides buffer-oriented stream concepts (`ReadStream`, `WriteStream`, `BufferSource`, `BufferSink`) with type-erased wrappers. Its documentation states: "It is not a networking library, yet it is the perfect foundation upon which networking libraries, or any libraries that perform I/O, may be built." Algorithms written against `any_stream` have "zero knowledge of Asio" and achieve "complete portability with no Asio dependency in algorithm code" (Boost.Capy (https://github.com/cppalliance/capy)).

Both Boost.HTTP and Boost.Capy are still in active development. The designs are early and may change. We are showing this work before it is finished because C++26 is about to ship, and the

window for reconsidering the execution model direction is closing. These libraries are presented not as finished proposals but as evidence that the approach is viable. The pattern they demonstrate, writing I/O algorithms against abstract buffer concepts independent of the transport, is what matters. The specific libraries are proof of concept, not the final answer.

The pattern is clear: standardize the buffer-oriented async I/O concepts and type-erased wrappers. Let the ecosystem provide the transports. The algorithms compose because they share the standard concepts, not because they share a specific socket implementation. This is a dramatically smaller standardization surface than either `std::execution` or full networking. And it solves the coordination problem: if everyone writes algorithms against the same buffer concepts, those algorithms interoperate regardless of the transport.

6. Conclusion

The committee recognized the need for a standard async model. That recognition was right. The question is one of priority.

GPU computing has CUDA, oneTBB, Kokkos, Taskflow, and stdexec on vcpkg. That ecosystem is thriving without the standard's help. Networking has no standard foundation at all. The tower of abstraction that every other language enjoys does not exist in C++.

The committee's energy is finite. Directing it toward the problem that only the standard can solve, an agreed-upon async I/O model that enables composable algorithms across library boundaries, would serve more C++ programmers than any other single investment the committee could make.

C++20 gave us coroutines. The building blocks are in the language. Twenty years of Asio practice and emerging libraries like Capy and IoAwaitables show that the design space is rich and worth exploring. The work is not finished, but the direction is clear.

The C++ Standard cannot connect to the Internet. It should.

References

1. Boost.Asio. https://www.boost.org/doc/libs/release/doc/html/boost_asio.html

2. Standalone Asio. https://github.com/chriskohlhoff/asio

3. folly::coro. Meta. https://github.com/facebook/folly/blob/main/folly/experimental/coro/README.md

4. Qt Signals and Slots. https://doc.qt.io/qt-6/signalsandslots.html

5. PEP 3156. "Asynchronous IO Support Rebooted: the asyncio Module." https://peps.python.org/pep-3156/

6. Django. https://github.com/django/django

7. TC39 Async/Await Proposal. https://tc39.es/proposal-async-await/

8. Express.js. https://github.com/expressjs/express

9. Next.js. https://github.com/vercel/next.js

10. Go net/http package. https://pkg.go.dev/net/http

11. Rust Blog. "Async-await on stable Rust!" 2019. https://blog.rust-lang.org/2019/11/07/Async-await-stable/

12. Tokio. https://github.com/tokio-rs/tokio

13. Axum. https://github.com/tokio-rs/axum

14. Spring Boot. https://github.com/spring-projects/spring-boot

15. C# async/await overview. https://dotnetcurry.com/csharp/869/async-await-csharp-dotnet

16. ASP.NET Core. https://github.com/dotnet/aspnetcore

17. Boost.Asio (GitHub). https://github.com/boostorg/asio

18. P2000R4. "Direction for ISO C++." WG21, 2022. https://open-std.org/jtc1/sc22/wg21/docs/papers/2022/p2000r4.pdf

19. nlohmann/json. https://github.com/nlohmann/json

20. cpp-httplib. https://github.com/yhirose/cpp-httplib

21. fmt. https://github.com/fmtlib/fmt

22. Lakos, John. *Large-Scale C++ Software Design*. Addison-Wesley, 1996. https://informit.com/store/large-scale-c-plus-plus-software-design-9780201633627

23. Boost.Beast. https://www.boost.org/doc/libs/release/libs/beast/

24. P3185R0. “A proposed direction for C++ Standard Networking based on IETF TAPS.” WG21. https://wg21.link/p3185r0

25. Networking TS reference implementation. Kohlhoff. https://github.com/chriskohlhoff/networking-ts-impl

26. P2453R0. “2021 October Library Evolution and Concurrency Networking and Executors Poll Outcomes.” WG21, 2022. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p2453r0.html

27. P2300R10. Dominiak et al. “std::execution.” WG21, 2024. https://open-std.org/jtc1/sc22/wg21/docs/papers/2024/p2300r10.html

28. stdexec. NVIDIA. https://nvidia.github.io/stdexec/

29. Burja, Samo. *Great Founder Theory.* 2020. https://www.samoburja.com/gft/

30. NVIDIA CUDA C/C++ Language Extensions. https://docs.nvidia.com/cuda/cuda-programming-guide/05-appendices/cpp-language-extensions.html

31. stdexec on vcpkg. https://vcpkg.link/ports/stdexec

32. P0912R5. “Merging Coroutines into C++20.” WG21. https://wg21.link/p0912r5

33. P3552. “Add a Coroutine Task Type.” WG21. https://wg21.link/p3552

34. D4007R0. Falco. “std::execution Needs More Time.” WG21, 2026. https://wg21.link/p4007

35. P3826R3. Niebler. “Fix Sender Algorithm Customization.” WG21, 2026. https://wg21.link/p3826

36. D4003. Falco et al. “IoAwaitables: A Coroutines-Only Execution Model.” WG21. https://wg21.link/p4003

37. TooManyCooks. https://github.com/tzcnt/TooManyCooks

38. Capy. https://github.com/cppalliance/capy

39. Stepanov. “The Standard Template Library.” 1994. https://stepanovpapers.com/Stepanov-The_Standard_Template_Library-1994.pdf

40. cppreference. “Allocator (named requirement).” https://en.cppreference.com/w/cpp/named_req/Allocator

41. cppreference. “RAII.” https://en.cppreference.com/w/cpp/language/raii

42. C++26 Working Draft, [exec] section. https://eel.is/c++draft/exec

43. Boost.HTTP. https://github.com/cppalliance/http