NetTS, ASIO and Sender Library Design Comparison

Draft Proposal

Document #:	P2471R1
Date:	2021-10-14
Project:	Programming Language C++
Audience:	LEWG Library Evolution SG1 Concurrency and Parallelism SG4 Networking

Reply-to: Kirk Shoop
<kirk.shoop@gmail.com>

Contents

1 Changes

1.1 R1

— added section response to P2471R0 (unnumbered paper)
— applied applicable corrections from the response paper to tables 2, 4, 5, 14, 16 & 17

1.2 R0

— first revision

2 Introduction

I have never seen the library designs of these libraries compared. This paper started as an email to the LEWG reflector. I was asked to make it a paper.

I used the following papers to fill in these tables:

— [N4771] "Working Draft, C++ Extensions for Networking"
— [P1322R3] "Networking TS enhancement to enable custom I/O executors"
— [P0958R3] "Networking TS changes to support proposed Executors TS"
— [P1943R0] "Networking TS changes to improve completion token flexibility and performance"
— [P2444R0] "The Asio asynchronous model"
— [P0443R14] "A Unified Executors Proposal for C++"
— [P2300R2] "std::execution"

I also searched in the ASIO repo

I split these out into vertical tables because horizontal scrolling sucks.

The library designs compared in tables below are:

— Asynchronous Operation design
— Initiating function design
— Algorithm design
— Associated Values design
— Executor design
— Execution context design

Within each section are three tables:

— [N4771] as it currently stands.
— [ASIO] as it currently stands.
— [P2300R2] as it currently stands

3 Notes

The theme that revealed itself to me while compiling these is that ASIO & NetTS use traits and partial-specialization vs. Sender/Receiver use concepts and CPOs.

There are other disparities in specific places, but the approach to library design traits/concepts and specialization/CPOs are the repeated differentiators I saw.

Another thing that these tables revealed is all the changes in the ASIO design in the last 2-3 years (after sender/receiver was proposed).

3.1 response to P2471R0 (unnumbered paper)

This section addresses the content of [response to P2471R0] "Response to P2471: Corrected NetTS, Asio, and Sender Library Design Comparison"

3.1.1 corrections

The corrections to Tables 2, 14, 16, and 17 are very helpful, thank you! These changes have been applied.

In tables 4 and 5, thank you for pointing out that it is convention to place the completion_token at the end. These changes have been applied.

3.1.2 conveying the whole design

In table 4, the deletion of the constructor arg is not due to it being incorrect, it just removes information that is specified in [N4771] from the table. I was attempting to make the composition of the token, handler and result clear. This deletion seems to hide this part of the spec.

I have altered table 4 in an attempt to make this clear.

3.1.3 convention ramifications

[response to P2471R0] is missing an explanation for why the completion token was last by convention, as well as the ramifications of putting it elsewhere. It turns out these are related.

The last arg by convention allows the function to not push arguments after the callback in the case it is a complicated bind expression or lambda

```
async_..(..., [...](...){
    ...
  });
```

The ramification of it preceding the trailing arg pack, in the case the function is a complicated expression like a lambda, is that the pack appears after the lambda.

```
async_..(..., [...](...){
    ...
  }, Arg...);
```

Allowing the initiation function to place the completion token anywhere in the arg list, makes any attempt at a concept definition for `initiation_function`, at best, challenging. A concept definition may not be possible.

3.1.4 heuristics for extracting an error

Completion signatures are defined by each initiating function. each initiating function can provide an error argument, conventionally the first argument. The type of that error is specified by each initiating function.

This arrangement allows partial completions where there is a non-success error value and valid, non-empty result value(s).

One result of this arrangement is that error and value types have to have default states in order to support errors to be delivered even when a valid result type cannot be constructed in this error condition, and also to support the separation of error-only and partial completion. For an error value type this default state would be Ok or Success (pet peeve - Ok and Success are not errors). For each value arg, the type must be constructible in an invalid or empty state. For types that do not have an empty or invalid state something like `std::optional<T>` would be used to add an empty state.

Another result of this arrangement is that completion tokens that translate from a completion signature to something that has an error channel (like `std::future`, coroutines, and senders), must use a heuristic to type-match the first arg (because there is no concept for error types to satisfy) and detect when the value of the first argument represents an error and preferably detects when the values are all valid and non-empty (again there is no concept for result values to detect an invalid or empty state) in order to send the error to the error channel for non-success error values and invalid result values (error-only) and alternatively, send the error + result to the value channel for non-success error values and valid result values (partial-completion).

Here are two possible implementations of `asSender` completion tokens that were supplied from ASIO. These demonstrate in code how these heuristics create many different implementations of `asSender`.

[as_sender_1.cpp (gist)] implementation of a completion token that maps to the following:

```
  as_sender:
    set_value(AllValues...)
    set_error(std::exception_ptr)
```

[as_sender_2.cpp (gist)] implementation of a completion token that maps to the following:

```
  as_sender:
    set_value(ResultValues...)
    set_error(std::error_code)
    set_error(std::exception_ptr)
```

This duplication for different heuristics applies to `use_awaitable` and `use_future` and any other transformation to something that has an error channel.

I asked a question in email that was not answered in the paper.

What is the ASIO take on the heuristics needed to map a completion signature to an error channel?

3.1.5 what are algorithms?

The changes to table 7 miss the mark completely.

In [P2444R0] composition is done by composing completion tokens lazily.

Example of lazy composition:

```
algoC(..., algoB(..., algoA(..., async_..(..., composition token),
 [...](...){
    ...
  })));
```

[P2444R0] & [N4771] support eager composition of completion tokens prior to calling the initiation function.

Example of eager composition:

```
async_..(..., algoA(..., algoB(..., algoC(..., [...](...){
    ...
  }))));
```

[response to P2471R0] redefines algorithm to mean something completely different from the definition in this paper. The redefinition is that any async initiating function can be an algorithm if it is implemented in terms of other initiating functions. I agree that this is a kind of algorithm, but there is no visible surface that can be used to create a concept. The fact that a initiating function is a composition of other initiating functions is an implementation detail and that implementation can change without affecting the initiating function's user facing surface or usage.

Trying to classify some async initiating functions as algorithms based on non-visible implementation details that change over time does not appear to me to be a useful classification and is certainly not what this paper intends an algorithm concept to represent.

> In this paper, an algorithm is something that wraps an existing async operation in a new async operation and can apply changes to the initiation and the completion of the existing async operation.

Both of the composition models in ASIO (lazy and eager) compose completion tokens. The only way to hook the initiation and completion of an async initiation function is to pass in a completion token

[N4771] cannot represent a lazy async operation and that is why it's only option is to wrap the completion token prior to calling the initiating function.

`deferred` composes lazily by packaging the values needed to initiate the initiating function that `deferred` was passed to, and producing a new initiating function that only takes a completion token. Each algorithm takes 0 or more initiating functions that each only take a completion token. The algorithm produces a new initiating function that only takes a completion token to allow the composition of another algorithm. Eventually a terminating completion token (`use_future`, a callback function - these are terminating) is passed to the initiating function returned by the last algorithm composed into the expression and that actually forwards to all the intervening packaged initiation functions until the original packaged initiating function is initiated and the final result is produced.

After defining algorithm one way (any initiating function that composes other async initiating functions), [response to P2471R0] adds another definition for algorithm - this time as a concept. The algorithm concept defined is `algorithm_over_packaged_operations`.

```
concept packaged_operation:
  (completion_token) -> Result;

concept algorithm_over_packaged_operations:
  (packaged_operation..., completion_token) -> Result;
```

> note I made one minor name change that seemed like a typo `packaged_ops...` -> `packaged_operation...`

`algorithm_over_packaged_operations` constrains the set of initiating functions that can be an algorithm to initiating functions that take zero or more `packaged_operations` followed by a completion token. This is an improvement, it means that `async_read` does not satisfy the concept of an algorithm and that matches my expectations. The concept does require lazy composition.

The pack is odd in that it means that different algorithms have a variable number of expected arguments - from 0 to Inf.. I am curious how this composition would work. Perhaps this is only intended to work when using lispy composition and is not supported with piping composition? I have supported this kind of thing before but it ends up splitting the first arg out of things like `when_all` to appear before the pipe and then have the rest as args to `when_all` after the pipe.

`packaged_operation` and `algorithm_over_packaged_operations` do not constrain the `Result`. This has the consequence that functions that return void, or an awaitable, or a sender would satisfy the concepts here, even though they terminated the composition of `packaged_operations`. Another option would be to change this concept to be specific to terminating the composition and use a different concept for an algorithm that constrains the result. Example:

```
concept packaged_operation:
  (completion_token) -> Result;

concept algorithm_over_packaged_operations:
  (packaged_operation..., completion_token) -> packaged_operation;

concept terminating_packaged_operation:
  !algorithm_over_packaged_operations;
  (packaged_operation..., completion_token) -> Result;
```

Keeping the `completion_token` as the last argument in an `algorithm_over_packaged_operations` limits these algorithms to a fixed number of `packaged_operation` arguments. The pack works fine for the concept not for an algorithm that wants to satisfy the concept. If an algorithm wanted to support a pack of `packaged_operation` arguments, then the `completion_token` argument must come before the pack. Any algorithm that did move the `completion_token` in front of the pack would not satisfy the concept as expressed. The solution is simple, but will probably incur unintended consequences in usage. Here is a solution:

```
concept packaged_operation:
  (completion_token) -> Result;

concept algorithm_over_packaged_operations:
  (completion_token, packaged_operation...) -> Result;
```

3.1.6 different names are not a correction

The changes to 8 also miss the mark completely, but in a different way.

[response to P2471R0] defines `packaged_operation`. This is great. This is not the only allowed composition model. Anyone can build their own completion token to initiate a different composition model and that composition model has no requirements to be compatible with the `packaged_operation`.

This is exactly the world the table in this paper depicted. As specific examples, `use_awaitable` and `use_sender` produce different composition models. ASIO itself supplies `use_awaitable`. In order for algorithms that satisfy `algorithm_over_packaged_operations` to compose with sender/receiver algorithms there would have to be preferably lossless conversion from `packaged_operation` to sender and sender to `packaged_operation`. The heuristics for extracting an error from a completion signature would prevent that conversion from being automatic. the user would have to be involved in deciding how to pack and extract the error and done signals into completion signatures.

Another thing left out of this section is that `packaged_operation` requires the lazy composition with argument moves that matches the sender/receiver model, but without the `operation_state` that allows the user to store the type in its own allocation or nest it in its own `operation_state`. There is a way for the user to supply storage for ASIO completion tokens, but there is a reason that code that does this has not been shown. There are three steps:

1. query for the size that will be allocated before calling the initiating function
2. create an allocator for that size in an existing allocation
3. associate that allocator with the completion handler for the initiating function

It would be useful to see the claim that no-alloc is possible in ASIO, be presented along with code that shows how to implement these steps.

3.1.7 associated values

The discussion of Associated values in [response to P2471R0] is incorrect. Sender/Receiver also allows defaults to be used, it, like everything else, is expressed as an algorithm. This algorithm is not specified in [P2300R2] and anyone can write it until it is added.

This algo to default a value would compose thusly

```
api() | with_default_query_value(get_allocator, allocator) | then...
```

this could be aliased to `with_default_allocator(allocator)`

3.1.8 Executor design

3.1.8.1 addressing stack exhaustion

Yes, default rescheduling each operation and default not rescheduling each operation, is a poor trade off. IMO both options are poor. The one good option that I know of that can prevent stack exhaustion is first-class tail-recursion in library or language (Lewis and I are experimenting with this in library, the current prototype runs `repeat_effect(just())` without stack exhaustion and without allocations and without scheduling and with cancellation - to the tune of about 1.6ns per repeat on my 2014 macbook pro - basically take this with a bag of salt for now).

ASIO has chosen to require that every async operation must schedule the completion on a scheduler (every read, every write, etc..). It is hard for me to have a sense of how many allocations and context switches that adds as a default.

sender/receiver has not decided to require that the completion be scheduled. If we did add that requirement to sender/receiver, then sender/receiver would still suffer from `inline_scheduler` being specified to an operation that completes synchronously, just as ASIO suffers from `inline_executor` today.

This is why I consider tail-call the only good solution. Scheduling solutions are all inferior (give thanks to Lewis for this shift in my understanding :)).

3.1.8.2 eager is required for performance

The only thing that I know of, that sender/receiver cannot currently represent with *full fidelity*, that the ASIO design does support is the option to eagerly start an operation before an api returns to the caller. The eager model in ASIO requires all composition to occur in the completion token before the api is called. Eager implementations, that allow attaching a continuation later, require allocation and synchronization overhead that would reverse the stated performance advantage. Using `deferred` is lazy composition and will behave exactly as sender does. The main resistance to using something like `std::execution::ensure_started(async_read(socket, buffer));` is the enormous cost of moving the socket and buffer handles into the sender and then allocating the `operation_state` and moving them again into the `operation_state` and linking that `operation_state` into the io queue without allocations vs. allocating an item in an io queue and depositing the args directly into that allocation. So same allocations and linking overhead, with an extra move of the args to the `operation_state`.

In three years of asking we have never been presented with code and a benchmark showing that the difference is measurable. I did produce a [benchmark for eager and lazy] that showed no significant difference and presented it to the executor paper's authors in September 2019 (two years ago). Again, no alternative benchmark was ever provided to show the problem that was claimed.

3.1.8.3 set_done/set_error force users to call them

as `done_as_error()` and `done_as_value()` indicate, it is not required for a cancelled operation to complete with `set_done()`. This is a recurring problem where the existence of a signal is characterized as a restriction on the implementor. This is C++. We don't do that. Just like you have the choice to return an `optional<T>` from a function if you might not be able to satisfy the post-conditions that are required to construct `T`, you have the option of calling `set_done()` instead of `set_value()`.

This is the same for `set_error()`. The un-nuanced statement is that - if you would not `throw` it, don't pass it to `set_error()`. As for all function design questions there is nuance here because `set_error` itself has no overhead of exception machinery, unless the error is `std::exception_ptr` or the sender is `co_awaited`, which would convert `set_error` to an exception.

One way to think of the receiver concept is as a representation of a function result of the type `expected<optional<variant<tuple<Vs...>...>>, variant<error_type...>>` without the overhead of packing and unpacking all the states in one value type.

— `set_error` maps to `expected<, variant<Es...>>` by being an overload set of functions that take one argument (`variant<>`)
— `set_done` maps to `optional<>`
— `set_value` maps to `variant<tuple<Vs...>...>` by being an overload set of functions (`variant<>`) where each function has a different set of arguments (`tuple<>`)

Essentially a receiver implements in library, `expected<>`, `optional<>`, `variant<>`, and `tuple<>` using pure language features. In other words, a receiver is as if all those types were built into the language.

3.1.8.4 which is a strict superset?

Neither ASIO or sender/receiver is a strict superset of the other. ASIO has eager initiation and sender/receiver has `set_error` and `set_done` with no heuristics to tease them out of an argument list (when arguments are present, ASIO executors can't pass arguments to the callback function - execute does not take a completion token - yet).

After years of communicating sender/receiver design and explaining what was missing from the ASIO design, ASIO has recently adopted many, but not all, of the sender/receiver features that were missing including lazy, overloaded completion functions, and cancellation.

All these were redesigned from scratch in ASIO over the past couple of years, with the stated goal of preventing major code changes for existing ASIO users. This goal does not feel like a goal that I would share as a member of the committee. It would depend heavily on what restrictions that goal imposed on the library design.

In addition to the compatibility for existing ASIO users, and the invention of a new cancellation type (`cancellation_slot`)instead of reusing `stop_token` from C++20.

The ASIO design relies on

— traits
— partial-specialization
— concrete types
— unspecified virtual interfaces between concrete types
— base classes

The sender/receiver design relies on

— concepts
— CPOs

IMO the question isn't which design is a subset or super-set, or if there is a way to adapt between different largely overlapping (after the recent changes in ASIO to increase the overlap) sets of functionality with vastly different designs.

IMO the question is which design **approach** is a good fit for the std library in 2021.

> The implementation and functionality in ASIO is first class. I am in awe of ASIOs utility and history and success. It is only the surface, if LEWG agrees with the feedback we have provided to ASIO for several years now, that would change to a design that fits in the standard library in 2021.

[P1322R3] proposes to add support for the standard library vendor to write multiple io context implementations, because Microsoft asked for a way to do this. [P1322R3] has not been applied to the NetTS yet. [P1322R3] does not allow users to write their own io context object that will work with `std::socket` as provided by the standard library they use, unless the user implements the unspecified virtual interface defined by each specific standard library implementation to interoperate with that specific standard libraries implementation of `std::socket`.

In contrast, a NetTS based on concepts and CPOs allows anyone to create types that can be used with each other. A socket type modeling a specified socket concept on epoll can be used with a different socket type modeling the same socket concept on uring. One socket type might be provided in a std library and the other socket type might be portably written by the user using only specified concepts.

3.1.8.5 Design changes

IMO a transform of the existing `io_context`, `socket`, etc.. design to one based on concepts and CPOs vs traits and types and unspecified interfaces would be straightforward (not trivial) and the result much cleaner. Some examples:

— Many of the member functions on the types today are in terms of other more fundamental functions. Moving those to CPOs and excluding them from the concepts makes writing your own `sockets` and `io_context` much less code.
— I would expect that the NetTS paper would shrink dramatically, because so many of the types become implementation details and do not need to be described (like services and base classes, etc..).
— I would expect a small header that maps the concepts to the types in ASIO would make ASIO a valid implementation of the redesigned NetTS.

I would be excited to finally be allowed to cooperate with the NetTS authors to change the surface without changing the implementation and functionality. IMO even eager is on the table, once actual code benchmarks are provided that demonstrate a measurable problem that is convincing to the committee - see (eager is required for performance).

4 Tables

4.1 Asynchronous Operation design

Table 1: NetTS - (N4771 is missing P1322, P0958, P1943, P2444)
13.2.7 Requirements on asynchronous operations

```
concept completion_token:
  async_result<
    completion_token,
    signature>
  ::completion_handler_type;

  async_result<
    completion_token,
    signature>
  ::return_type;
```

```
concept signature:
  (ErrorsAndValues...) -> void;

concept completion_handler_type:
  constructible<
    completion_handler_type,
    completion_token>;
  invocable<
    completion_handler_type,
    signature>;

concept result_type:
  constructible<
    result_type,
    completion_handler_type>;
```

Table 2: ASIO - (ASIO has P1322, P0958, P1943, P2444)

```
concept completion_token:
  async_result<
    completion_token,
    signature...>
  ::initiate(
    initiation,
    completion_token,
    Args...) -> Result;
```

```
concept signature:
  (ErrorsAndValues...) -> void;

concept initiation:
  (completion_handler, Args...) -> void;

concept completion_handler:
  invocable<
    completion_handler,
    signature>...;
```

Table 3: Sender/Receiver - (P2300)

```
concept sender:
  connect(sender, receiver) -> operation_state;

concept operation_state:
  start(operation_state) -> void;

concept receiver:
  set_value(receiver, Values...) -> void;
  set_error(receiver, Error) -> void;
  set_done(receiver) -> void;
```

4.2 Initiating function design

Table 4: NetTS - (N4771 is missing P1322, P0958, P1943, P2444)
13.2.7 Requirements on asynchronous operations

```
Any function that takes a completion_token, by convention, asthe last
argument, and returns:

(..., completion_token)
  -> decltype(async_result<
    completion_token,
    signature>::result_type
      // for clarity on compostion
      // result_type(
      //   async_result<
      //     completion_token,
      //     signature>
      //       ::completion_handler_type(
      //         completion_token))
    );

Since completion token placement is a convention, one could alsoput
the completion token in front of a variadic argument list.
The ramifications of this are addressed inconvention ramifications
```

Table 5: ASIO - (ASIO has P1322, P0958, P1943, P2444)

```
Any function that takes a completion_token, by convention, asthe last
argument, and returns the result of:

(..., completion_token)
  -> decltype(async_result<
    completion_token,
    signature...>
  ::initiate(
    initiation,
    completion_token,
    Args...));

Since completion token placement is a convention, one could alsoput
the completion token in front of a variadic argument list.
The ramifications of this are addressed inconvention ramifications
```

Table 6: Sender/Receiver - (P2300)

```
Any function returning a sender

(...) -> sender;
```

4.3 Algorithm design

Table 7: NetTS - (N4771 is missing P1322, P0958, P1943, P2444)

```
Unspecified, but without async_initiate the only option
I know of is eager composition of completion tokens before
calling the initiating function:

concept algorithm:
  (completion_token) -> completion_token;
```

Table 8: ASIO - (ASIO has P1322, P0958, P1943, P2444)

```
Includes the above and:

Any specific completion_token can define a new
composable_type and return that.

The deferred completion token (July 2021) isan example of this.
Other examples include the use_awaitable anduse_sender completion tokens.

One of the infinite possible shapes for that new
composable_type could be:

concept algorithm
  (composable_type) -> composable_type;

concept composable_type:
  (completion_token) -> Result;

The shape defined in P2471R0response is another alternative

concept packaged_operation:
  (completion_token) -> Result;

Algorithms over packaged asynchronous operations may look like:

concept algorithm_over_packaged_operations:
(packaged_operation..., completion_token) -> Result;
```

Table 9: Sender/Receiver - (P2300)

```
concept algorithm:
  (sender) -> sender
```

4.4 Associated values design

Table 10: NetTS - (N4771 is missing P1322, P0958, P1943, P2444)
13.2.2 Executor requirements

```
concept associated_executor:
  associated_executor<Source, Default>::type;
  associated_executor<Source, Default>
    ::get(source, default) -> executor; // static

concept associated_allocator:
  associated_allocator<Source, Default>::type;
  associated_allocator<Source, Default>
    ::get(source, default) -> allocator; // static
```

Table 11: ASIO - (ASIO has P1322, P0958, P1943, P2444)

```
Includes the above and:

concept associated_cancellation_slot:
  associated_cancellation_slot<Source, Default>::type;
  associated_cancellation_slot<Source, Default>
    ::get(source, default) -> cancellation_slot; // static
```

Table 12: Sender/Receiver - (P2300)

```
concept scheduler_provider:
  get_scheduler(scheduler_provider) -> scheduler;

concept allocator_provider:
  get_allocator(allocator_provider) -> allocator;

concept stop_token_provider:
  get_stop_token(stop_token_provider) -> stop_token;
```

4.5 Executor design

Table 13: NetTS - (N4771 is missing P1322, P0958, P1943, P2444)
13.2.2 Executor requirements

```
concept executor:
  executor::context() -> execution_context;
  executor::on_work_started() -> void;
  executor::on_work_finished() -> void;
  executor::dispatch(()->void, Allocator) -> void;
  executor::post(()->void, Allocator) -> void;
  executor::defer(()->void, Allocator) -> void;
```

Table 14: ASIO - (ASIO has P1322, P0958, P1943, P2444)

```
concept executor:
  execute(executor, ()->void) -> void;

Uses properties to support the functionality from the tableabove
```

Table 15: Sender/Receiver - (P2300)

```
concept scheduler:
  schedule(scheduler) -> sender;
```

4.6 Execution Context design

Table 16: NetTS - (N4771 is missing P1322, P0958, P1943, P2444)
13.2.3 Execution context requirements

```
concept execution_context:
  is_base_of<net::execution_context, execution_context>;
  execution_context::executor_type;
  execution_context::get_executor()
    -> execution_context::executor_type;
```

Table 17: ASIO - (ASIO has P1322, P0958, P1943, P2444)

```
concept execution_context:
  is_base_of<net::execution_context, execution_context>;
  execution_context::executor_type;
  execution_context::get_executor()
    -> execution_context::executor_type;
```

Table 18: Sender/Receiver - (P2300)

```
concept execution_context:
  no-requirements
```

5 References

[ASIO] Christopher Kohlhoff. ASIO source (github).
https://github.com/chriskohlhoff/asio

[as_sender_1.cpp (gist)] Christopher Kohlhoff. as_sender_1.cpp.
https://gist.github.com/chriskohlhoff/cb8b4719890e7769cc759f73eacb3496

[as_sender_2.cpp (gist)] Christopher Kohlhoff. as_sender_2.cpp.
https://gist.github.com/chriskohlhoff/c1da9c5f9d2adcb2b94d27beb0880739

[benchmark for eager and lazy] Kirk Shoop. benchmark for eager and lazy.
https://quick-bench.com/q/NXMiRTZso1fPUiptmvRSPUutuPQ

[N4771] Jonathan Wakely. 2018-10-08. Working Draft, C++ Extensions for Networking.
https://wg21.link/n4771

[P0443R14] Jared Hoberock, Michael Garland, Chris Kohlhoff, Chris Mysen, H. Carter Edwards, Gordon Brown, D. S. Hollman. 2020-09-15. A Unified Executors Proposal for C++.
https://wg21.link/p0443r14

[P0958R3] Christopher Kohlhoff. 2021-03-15. Networking TS changes to support proposed Executors TS.
https://wg21.link/p0958r3

[P1322R3] Christopher Kohlhoff. 2021-02-15. Networking TS enhancement to enable custom I/O executors.
https://wg21.link/p1322r3

[P1943R0] Christopher Kohlhoff. 2019-10-07. Networking TS changes to improve completion token flexibility and performance.
https://wg21.link/p1943r0

[P2300R2] Michał Dominiak, Lewis Baker, Lee Howes, Kirk Shoop, Michael Garland, Eric Niebler, and Bryce Adelstein Lelbach. std::execution.
https://isocpp.org/files/papers/P2300R2

[P2444R0] Christopher Kohlhoff. 2021-09-15. The Asio asynchronous model.
https://wg21.link/p2444r0

[response to P2471R0] Jamie Allsop, Christopher Kohlhoff, and Klemens Morgenstern. Response to P2471: Corrected NetTS, Asio, and Sender Library Design Comparison. https://wiki.edg.com/pub/Wg21telecons2021/LibraryEvolutionWorkingGroup/Response_to_P2471.pdf