Document Number: P2469r0
Date: 2021-10-04
Project: Programming Language C++
Audience: WG21, LEWG
Authors: Jamie Allsop
Vinnie Falco
Richard Hodges
Christopher Kohlhoff
Klemens Morgenstern
Reply-to: Christopher Kohlhoff <chris@kohlhoff.com>

Response to P2464: The Networking TS is baked, P2300 Sender/Receiver is not.

Abstract

P2464 draws a number of inaccurate conclusions about the asynchronous model described in the Networking TS ("Asio/Net.TS") and how it relates to P2300 Sender/Receiver ("P2300"). To assist readers confused by P2464, this paper will help set the record straight and address those inaccuracies and misconceptions. In short:

- The executor is an essential part of any asynchronous model. A model that lacks an equivalent customisation point for asynchronous "tail calls" is fundamentally unsuited to domains in which C++ is extensively used.
- Asio/Net.TS has a well defined vocabulary for composition based around the *completion token*.
- The asynchronous model of Asio/Net.TS has also evolved to support new use cases while also being careful not to leave existing use cases behind, and the strength of the composition model is testament to that. The model is the result of growth and adaptation from use in the real world, and is one reason it is so widely deployed.
- P2300 does not propose an asynchronous model, it proposes a DSL for composing asynchronous operations which assumes an asynchronous model.
- The asynchronous model in the Asio/Net.TS offers a superset of the functionality exposed through P2300.
- Asio/Net.TS is exactly the foundational, scalable asynchronous model upon which the higher level abstractions proposed in P2300 can and should be layered.
- Put simply, the DSL proposed by P2300 does not enable any missing functionality, and could be added in the future, after evolution to address support for a wider variety of use cases.

How did we get here?

The final conclusions that P2464 puts forward do a great job of highlighting how the committee process's priorities seem to have gone awry in recent times — favouring standardisation of new, minimally used ideas over well established practice. The kind of established practice that has been under constant challenge and evolution as needed to remain relevant, while offering best of breed functionality to the widest possible audience.

With such exposure we can all agree such offerings are never "perfect" and so lately we seem to prefer favouring the elusive type of perfection that is only achievable with ideas and not actual implementations, where reality bites and bites hard.

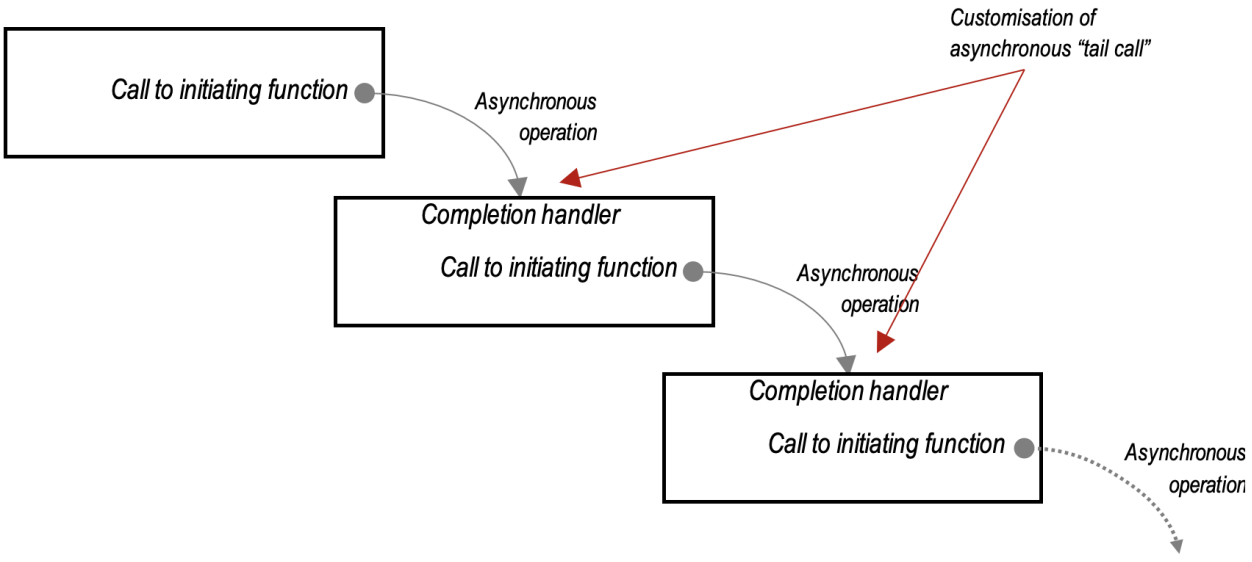
There are always trade-offs, and in standardising C++ as a committee we have always laboured to ensure that, at a minimum we attempt to either standardise essential functionality (like networking and threads), or further the core building blocks so that domain level trade-offs can be decided by users building on that foundation.

The Networking TS is now six-years-old and would appear to possibly become exactly such a casualty of this ever increasing and unhealthy trend towards standardising unchallengeable ideas over practice. This is not the first paper to call this out and sadly we expect it will not be the last.

What is an executor in Asio/Net.TS?

The *executor* is the *customisation point for the "tail call" of an asynchronous operation*.

Specifically, when an asynchronous operation completes, it must deliver its result to a continuation, called a *completion handler* in Asio/Net.TS. At this point, the asynchronous operation ceases to exist. It is analogous to a synchronous operation returning from a function call.



However, unlike their synchronous counterparts, asynchronous operations are control flows that may outlive the synchronous confines of threads and block scopes. A customisation point at each “tail call” is required for:

- **Correctness.** For example, to coordinate access to resources that are shared with other asynchronous operations, or other parts of the application.
- **Fairness.** A naïve asynchronous operation implementation that permits reentrant completion can lead to starvation of other tasks or, worse, stack overflow. The “tail call” customisation point allows us to manage immediate and delayed completions, to ensure that a single chain of operations does not monopolise an event loop.
- **Performance**. Techniques for maximising throughput often involve executing related work close together, in both space and time.

Importantly, expressing this facility as a customisation point allows these requirements to be managed as a cross-cutting concern to the asynchronous operation itself. We can write our asynchronous algorithms once, as a composition of underlying operations, and allow the user to make the correct trade-offs for their use case.

Error handling within the “tail call”

After a synchronous function call returns, any errors that subsequently arise are obviously somebody else’s problem. Similarly, by definition, any errors that occur following the asynchronous “tail call” do not return back to the asynchronous operation, as it has ceased to exist, just like the scope of the analogous synchronous function call that we returned from. Consequently, the “tail call” customisation point relates only to submission.

How can we handle continuation on something potentially failing? We simply adapt our potentially-failing interface to the “tail call” customisation point requirements. For example, a use case that requires more complex and explicit handling of execution errors (e.g. a remote code executor) can implement the executor’s `post` and `dispatch` functions in a way that looks similar to this pseudo-code:

```
struct remote_code_executor
{
  void submit(auto f, auto on_failure);
  auto bind_error_handler(on_failure of) -> bound_custom_executor;
};

struct bound_remote_code_executor
{
  Remote_code_executor inner_ex_;
  on_failure on_failure_;
  void post(f) { inner_ex_.submit(f, on_failure_); }
  void dispatch(f) { inner_ex_.submit(f, on_failure_); }
  // ...
};

auto run_async_op(remote_code_executor ce, on_failure err, CompletionToken&&
token)
{
```

```
  return async_op(bind_executor(ce.bind_error_handler(err), token));
}
```

A history of executors in Asio

The need for an asynchronous “tail call” customisation point was identified very early in Asio’s development. In early 2006, the `asio_handler_dispatch` ADL customisation point was added to address these requirements, and then later renamed to `asio_handler_invoke`.

With the arrival of C++11 it was found that the `asio_handler_invoke` approach was incompatible with move-only semantics, and in particular with move-only completion handlers. To solve this it was necessary to decouple the policy object (now called the executor) from the customisation point itself (`associated_executor`).

The executor-as-policy-object also acquired additional capabilities at this time, such as the ability to distinguish between immediate and delayed completion. These capabilities enabled the customisation point to be used generically with a diverse range of asynchronous event sources. This form was also eventually specified in the Networking TS.

From around 2014, SG1 found itself presented with multiple proposals called “executor”, and eventually instructed the various authors to find a unified model that addressed all concerns. As the requirements of an asynchronous “tail call” customisation point differed from other uses, this ultimately resulted in the P0443 form of an executor, where these requirements were expressed as optional functionality, via properties.

We now have well over a year of production experience in Asio with the P0443 form of an executor. Functionally, the vast majority of users have been unaffected, as they rarely interact with executors except to specify them as the desired policy. For those who do, there has been some reduction in usability, due to the increased complexity of the unified interface.

Related existing practice

We can see the same or similar customisation points, with an equivalent executor-as-policy-object, in other successful asynchronous models, including:

- Grand Central Dispatch, as "DispatchQueue"
- Swift structured concurrency, as "Executor"
- Java Netty library, as "EventExecutor"
- .NET, as "SynchronizationContext"

While they differ in the details, this convergent evolution is indicative of the underlying requirement driving the design of asynchronous models.

What happens if we don't have executors?

Consider the following snippet from a libunifex example. The code shown here repeatedly reads bytes from a pipe until a number of seconds have elapsed.

```
auto pipe_bench = [](auto& rPipeRef, auto& buffer, auto scheduler, int seconds,
                     [[maybe_unused]] auto& data, auto& reps, [[maybe_unused]] auto& offset) {
  return defer([&, scheduler, seconds] {
    return defer([&] {
      return
        // do read:
        async_read_some(rPipeRef, as_writable_bytes(span{buffer.data() + 0, 1}))
        | discard
        | then([&] {
            UNIFEX_ASSERT(data[(reps + offset) % sizeof(data)] == buffer[0]);
            ++reps;
          });
    })
      | typed_via(scheduler)
        // Repeat the reads:
      | repeat_effect()
        // stop reads after requested time
      | stop_when(schedule_at(scheduler, now(scheduler) + std::chrono::seconds(seconds)))
        // complete with void when requested time expires
      | let_done([]{return just();});
  });
};
```

As specified in P2300, sender operations such as `async_read_some` are permitted to complete with a reentrant call to the receiver. Assuming an always-ready pipe, the only thing that stops this from causing stack overflow is the use of the `typed_via` algorithm. If we remove the `typed_via`, an always-ready pipe results in unbounded recursion.

This puts the onus on the user to insert a `typed_via`, or equivalent, between asynchronous operations, to break the asynchronous loop. Particularly when writing network-facing code, it may not always be immediately obvious to the inexperienced asynchronous programmer that such an always-ready condition is possible.

Furthermore, requiring this to be handled explicitly, as above, introduces additional complexity when composing asynchronous operations as layers of abstraction. If instead of `async_read_some` were using an HTTP read operation, any loops within that would also need an explicit `typed_via`. And so on, and so on, down through each layer.

Nor does the use of `typed_via` here solve the reentrancy issue in a generic way. The scheduler object that is passed in could, in turn, be implemented as a direct call (e.g. an inline scheduler) or as a potentially direct call (e.g. a thread pool scheduler when the caller is already in the pool). This code is only known to work because of assumptions about the behaviour of the concrete scheduler being used.

In order to solve this generically, the algorithm must either "reach through the generic scheduler concept" to determine if it has the required concrete behaviour, or explicitly use a different scheduler that is known to provide the necessary guarantee, such as the trampoline scheduler used in some other libunifex examples. This prevents the algorithm author from writing algorithms that treat their scheduling requirements as a cross cutting concern, rendering it impossible to write generic algorithms.

This is user-hostile as default behaviour.

It may be tempting to think that this can be addressed by simply mandating that senders are not allowed to complete reentrantly. However, without also making this a customisation point, all operations are pessimised. This will especially affect the performance of composed operations negatively, since they are unable to optimize internal completion steps, causing performance degradation at scale. Knowing the context in which a particular asynchronous operation will be used allows the user to select the appropriate scheduling strategy, such as favouring throughput over fairness, or vice versa.

In contrast, the Asio/Net.TS model allows us to make this choice independent of the implementation of the asynchronous operation, at a higher level. Composed operations can implement generic algorithms and let the user pick those characteristics.

This shows that P2300 is not a complete asynchronous model, but rather a DSL for composing and creating graphs of operations. It does not specify everything that is required for correct, safe-by-default use of asynchronicity, and relies on the user to take care when structuring their code around other senders. We can see how this is evident when we modified the example above and triggered a stack overflow, or in the echo server example shown below. These examples make assumptions about the implementation details of the specific I/O operations being used.

Asio/Net.TS is the low-level foundation for P2300

The Asio/Net.TS model is a foundational model, on which DSLs such as P2300 can be implemented. In particular:

- The Asio/Net.TS model exposes higher order asynchronous operations. This allows them to produce senders (typed senders, in fact), in addition to eager operations, coroutines, fibers, etc, from a single implementation.
- The Asio/Net.TS model incorporates executors to ensure correct, safe-by-default asynchronicity. These can be ignored in use cases where they need not apply (as is apparently the case for the use cases supported by P2300).

The following examples illustrate and compare Asio/Net.TS use to P2300:

echo server	Simple echo server using a lazy task graph DSL
inclusive scan	Asynchronous computation
GPU integration	Incorporates GPU computation into asynchronous flow
recursive file copy	Illustrates the use of structure concurrency techniques
`then` algorithm	A generic algorithm to attach a transformation
`retry` algorithm	A generic algorithm to retry operations on error

Example: echo server

Many existing users of Asio prefer to express their asynchronous control flow using composition mechanisms such as coroutines, fibers, or simple chaining of callbacks. However, construction of asynchronous compositions as a lazy task graph DSL, similar to P2300, is already supported in the Asio/Net.TS model. This example illustrates how an echo server may be written using this approach, with adapters like `on_success` and algorithms like `repeat` (which is much like the retry example shown below).

Asio/Net.TS	**P2300**
`template <std::size_t N, class CompletionToken>` `auto echo(tcp::socket& sock, char (&buf)[N], CompletionToken&& token)` `{` `  return repeat(` `      sock.async_read_some(buffer(buf), deferred)` `        | on_success(` `            [&sock, buf](std::size_t n) {` `              return async_write(sock, buffer(buf, n), deferred);` `            }` `          ),` `      std::forward<CompletionToken>(token)` `    );` `}`	`outstanding.start(` `    EX::repeat_effect_until(` `          EX::let_value(` `              NN::async_read_some(ptr->d_socket,` `                                  context.scheduler(),` `                                  NN::buffer(ptr->d_buffer))` `        | EX::then([ptr](::std::size_t n){` `            ::std::cout << "read='" << ::std::string_view(ptr->d_buffer, n) << "'\n";` `            ptr->d_done = n == 0;` `            return n;` `        }),` `          [&context, ptr](::std::size_t n){` `            return NN::async_write_some(ptr->d_socket,` `                                        context.scheduler(),` `                                        NN::buffer(ptr->d_buffer, n));` `          })` `        | EX::then([](auto&&...){})` `        , [owner = ::std::move(owner)]{ return owner->d_done; }` `    )` `);`
Discussion	
This code provides a safe default in that, even if the read or write operations complete immediately, the users code is not run reentrantly. This ensures fairness between connections, and mitigates the risk of stack overflow.	This code makes an assumption about the specific behaviour of the read and write operations' implementations, namely that they will not invoke the receiver reentrantly even if there is an immediate completion.

Example: inclusive scan

The following example shows the asynchronous computation of an inclusive scan. The implementation of the Asio/Net.TS version uses coroutines, although it is still exposed as a generic, reusable asynchronous operation.

Asio/Net.TS

```
awaitable<std::vector<double>>
async_inclusive_scan_impl(std::span<const double> input,                  // 1
                          std::span<double> output,                       // 1
                          double init,                                    // 1
                          std::size_t tile_count)                         // 3
{
  std::size_t const tile_size = (input.size() + tile_count - 1) / tile_count;

  std::vector<double> partials(tile_count + 1);                           // 4
  partials[0] = init;                                                     // 4

  co_await bulk(co_await this_coro::executor, tile_count,
      [&](std::size_t i)
      {                                                                   // 7
        auto start = i * tile_size;                                       // 8
        auto end   = std::min(input.size(), (i + 1) * tile_size);         // 8
        partials[i + 1] = *--std::inclusive_scan(begin(input) + start,    // 9
                                                 begin(input) + end,      // 9
                                                 begin(output) + start);  // 9
      }, use_awaitable);

  std::inclusive_scan(begin(partials), end(partials),                     // 12
                      begin(partials));                                   // 12

  co_await bulk(co_await this_coro::executor, tile_count,
      [&](std::size_t i)
      {
        auto start = i * tile_size;                                       // 14
        auto end   = std::min(input.size(), (i + 1) * tile_size);         // 14
        std::for_each(output.begin() + start, output.begin() + end,       // 14
          [&] (double& e) { e = partials[i] + e; }                        // 14
        );
      }, use_awaitable);

  co_return partials;                                                     // 15
}

template<class Executor, class CompletionToken>
auto async_inclusive_scan(Executor&& exec,
                          std::span<const double> input,                  // 1
```

P2300

```
sender auto async_inclusive_scan(scheduler auto sch,                      // 2
                                 std::span<const double> input,           // 1
                                 std::span<double> output,                // 1
                                 double init,                             // 1
                                 std::size_t tile_count)                  // 3
{
  std::size_t const tile_size = (input.size() + tile_count - 1) / tile_count;

  std::vector<double> partials(tile_count + 1);                           // 4
  partials[0] = init;                                                     // 4

  return transfer_just(sch, std::move(partials))                          // 5
       | bulk(tile_count,                                                 // 6
          [=](std::size_t i, std::vector<double>& partials) {             // 7
            auto start = i * tile_size;                                   // 8
            auto end   = std::min(input.size(), (i + 1) * tile_size);     // 8
            partials[i + 1] = *--std::inclusive_scan(begin(input) + start,// 9
                                                     begin(input) + end,  // 9
                                                     begin(output) + start); // 9
          })                                                              // 10
       | then(                                                            // 11
          [](std::vector<double>& partials) {
            std::inclusive_scan(begin(partials), end(partials),           // 12
                                begin(partials));                         // 12
            return std::move(partials);                                   // 13
          })
       | bulk(tile_count,                                                 // 14
          [=](std::size_t i, std::vector<double>& partials) {             // 14
            auto start = i * tile_size;                                   // 14
            auto end   = std::min(input.size(), (i + 1) * tile_size);     // 14
            std::for_each(output + start, output + end,                   // 14
              [&] (double& e) { e = partials[i] + e; }                    // 14
            );
          })
       | then(                                                            // 15
          [=](std::vector<double>& partials) {                            // 15
            return output;                                                // 15
          });                                                             // 15
}
```

```
                        std::span<double> output,                       // 1
                        double init,                                    // 1
                        std::size_t tile_count,
                        CompletionToken&& token)
{
  return co_spawn(std::forward<Executor>(exec),
                  async_inclusive_scan_impl(input, output, init, tile_count),
                  std::forward<CompletionToken>(token));
}
```

This algorithm is implemented in terms of a generic `bulk` asynchronous operation. This algorithm is not currently part of Asio, so a possible implementation is shown below.

Asio/Net.TS

```
template <class F>
awaitable<void> bulk_impl(std::size_t first, std::size_t last, F f)
{
  auto n = last - first;
  if (n == 1)
  {
    f(first);
  }
  else if (n > 1)
  {
    co_await (
        bulk_impl(first, first + n / 2, f)
          && bulk_impl(first + n / 2, last, f)
      );
  }
}

template <class Executor, class F, class CompletionToken>
auto bulk(Executor&& exec, std::size_t n, F f, CompletionToken&& token)
{
  return co_spawn(std::forward<Executor>(exec),
                  [n, f]{ return bulk_impl(0, n, f); },
                  std::forward<CompletionToken>(token));
}
```

Example: GPU integration

This example demonstrates a simple server that incorporates GPU computation as asynchronous operations. This shows how entities that would be considered “schedulers” in P2300 are presented as classes with asynchronous operations under the Asio/Net.TS model. Here it is wrapped as a `cuda_context`, having a single asynchronous operation `async_compute`.

Asio/Net.TS

```
struct checksum_chunks_impl
{
  std::size_t num_chunks;
  std::size_t chunk_size;
  const unsigned char* data;
  unsigned int* results;

  __device__ void operator()(std::size_t x, std::size_t y)
  {
    std::size_t start = x * chunk_size;
    std::size_t end = start + chunk_size;
    results[x] = 0;
    for (std::size_t i = start; i < end; ++i)
      results[x] = (results[x] + data[i]) % 256;
  }
};

template <class CompletionToken>
auto checksum_chunks(cuda_context& ctx, const cuda_device_vector<unsigned char>& data,
    cuda_device_vector<unsigned int>& results, CompletionToken&& token)
{
  checksum_chunks_impl impl =
  {
    .num_chunks = results.size(),
    .chunk_size = data.size() / results.size(),
    .data = data.data(),
    .results = results.data()
  };
  return bulk(ctx, impl, results.size(), 1, std::forward<CompletionToken>(token));
}

struct reduce_results_impl
{
  std::size_t num_chunks;
  unsigned int* results;

  __device__ void operator()(std::size_t x, std::size_t y)
  {
```

```
    unsigned int result = 0;
    for (std::size_t i = 0; i < num_chunks; ++i)
      result = (result + results[i]) % 256;
    results[0] = result;
  }
};

template <class CompletionToken>
auto reduce_results(cuda_context& ctx, cuda_device_vector<unsigned int>& results, CompletionToken&& token)
{
  reduce_results_impl impl = { .num_chunks = results.size(), .results = results.data() };
  return bulk(ctx, impl, 1, 1, std::forward<CompletionToken>(token));
}

void checksum(tcp::socket s, yield_context yield)
{
  try
  {
    constexpr std::size_t chunk_size = 1024 * 1024;
    constexpr std::size_t num_chunks = 16;
    constexpr std::size_t message_size = num_chunks * chunk_size;

    cuda_context cuda_ctx(s.get_executor());
    std::vector<unsigned char> data(message_size);
    std::vector<unsigned int> results(num_chunks);
    cuda_device_vector<unsigned char> device_data(message_size);
    cuda_device_vector<unsigned int> device_results(num_chunks);

    for (;;)
    {
      async_read(s, buffer(data), yield);
      make_linked_group(
          copy(cuda_ctx, data.begin(), data.end(), device_data.begin(), deferred),
          checksum_chunks(cuda_ctx, device_data, device_results, deferred),
          reduce_results(cuda_ctx, device_results, deferred),
          copy(cuda_ctx, device_results.begin(), device_results.end(), results.begin(), deferred)
        ).async_wait(yield);
      async_write(s, buffer(std::to_string(static_cast<unsigned int>(results[0])) + "\n"), yield);
    }
  }
  catch (const std::exception&)
  {
  }
}
```

It is worth noting that, as an optimisation, this example submits the task graph to the CPU without waiting for the intermediate responses. It also uses Boost.Coroutine due to the lack of C++20 coroutine support with nvcc.

Example: recursive file copy

This example illustrates the use of structured concurrency techniques to manage copying a directory tree. The first snippets shown here demonstrate the composition of read and write operations into a simple linear sequence, to copy bytes from the source file to the destination.

Asio/Net.TS

```
mutable_buffer align(mutable_buffer buf)
{
  void* data = buf.data();
  std::size_t size = buf.size();
  if (std::align(buf_align, buf_size, data, size) == nullptr)
    std::abort();
  return mutable_buffer(data, size);
}

awaitable<std::size_t> async_copy_file(const fs::path& from, const fs::path& to)
{
  stream_file from_file = open_file_read_only(co_await this_coro::executor, from);
  stream_file to_file = open_file_write_only(co_await this_coro::executor, to);

  std::vector<std::byte> buf_space(buf_size + buf_align);
  auto buf = align(buffer(buf_space));

  std::size_t bytes_copied = 0;
  while (true)
  {
    auto [e, n] = co_await from_file.async_read_some(buf,
as_tuple(use_awaitable));
    if (e == stream_errc::eof) break;
    if (e) throw std::system_error(e);
    co_await async_write(to_file, buffer(buf, n), use_awaitable);
    bytes_copied += n;
  }
  co_return bytes_copied;
}
```

P2300

```
auto copy_file(io_uring_context::scheduler s, const fs::path& from, const fs::path& to) {
  // introduce new async scope for
  // index_, buffer_, repeat_, from_, to_
  return let_value_with(
    [s, from = from.string(), to = to.string()] {
      // open the from and to files and store the
      // handles in the scope
      return std::make_tuple(
        open_file_read_only(s, from),
        open_file_write_only(s, to)
      );
    },
    [
      // define state across loop iterations
      index_ = size_t{0},
      buffer_ = std::vector<char>{},
      repeat_ = true
    ](auto& state) mutable {
      // reference the file handles
      auto& [from_, to_] = state;

      // set buffer size
      buffer_.resize(bufferSize + bufferAlign);
      buffer_.resize(buffer_.capacity());

      // align buffer
      void* lvalueBegin = (void*)buffer_.data();
      size_t lvalueSize = buffer_.size();
      if (nullptr == std::align(bufferAlign, bufferSize, lvalueBegin, lvalueSize)) {
        std::abort();
      }
      span<char> buffer{(char*)lvalueBegin, lvalueSize};

      // read and write loop
      return read_some_write_all(from_, to_, buffer, index_, repeat_)
      // update index_
      | then([&](auto bytesWritten){
        index_ += bytesWritten;
      })
      | repeat_effect_until([&]{return !repeat_;})
```

	`` // result is total number of bytes copied | then([&]{ return index_; }); }); } ```

This second snippet shows the use of structured concurrency to manage “fan out” as the work is created, and “fan in” as the results are collected.

Asio/Net.TS	**P2300**
`` awaitable<std::size_t> copy_files(const fs::path& from, std::vector<fs::directory_entry>::iterator first, std::vector<fs::directory_entry>::iterator last, const fs::path& to, steady_timer& turn_timer, std::size_t& active_copies) { auto n = last - first; if (n == 1) co_return co_await queue_file_copy(from, *first, to, turn_timer, active_copies); else if (n > 1) { auto [n1, n2] = co_await (copy_files(from, first, first + n / 2, to, turn_timer, active_copies) && copy_files(from, first + n / 2, last, to, turn_timer, active_copies)); co_return n1 + n2; } else co_return 0; } awaitable<std::size_t> copy_files(const fs::path& from, const fs::path& to) { steady_timer turn_timer(co_await this_coro::executor); turn_timer.expires_at(steady_timer::time_point::max()); std::size_t active_copies = 0; std::vector<fs::directory_entry> entries{ fs::recursive_directory_iterator(from), fs::recursive_directory_iterator()}; co_return co_await copy_files(from, entries.begin(), entries.end(), to, turn_timer, active_copies); } ```	``` auto copy_files(io_uring_context::scheduler s, const fs::path& from, const fs::path& to) noexcept { // some of this state cannot be moved or copied // create a type that allows them to be constructed in place using state_t = std::tuple< std::atomic<size_t>, fs::recursive_directory_iterator, std::atomic<int>, unifex::async_manual_reset_event>; // create new async scope for // s_, from_, to_, bytesCopied_, entry_, pending_, drain_ return let_value_with([]() -> state_t {return {};}, // store state in scope [s_ = s, from_ = from.string(), to_ = to.string()](state_t& state) { // reference the state auto& [bytesCopied_, entry_, pending_, drain_] = state; // initialize the state entry_ = fs::recursive_directory_iterator(from_); drain_.set(); // loop through all the directory entries and copy all the files return establish_scope([&](unifex::async_scope& scope_) { return sequence(limit_open_files(pending_, drain_), queue_file_copy(s_, from_, to_, entry_, scope_, bytesCopied_, pending_, drain_)) 	repeat_effect_until([&]{return entry_ == end(entry_) || ++entry_ == end(entry_);}); }) // the result is the count of bytes that were copied | then([&]{return bytesCopied_.load();}); }); } ```

Example algorithm: then

This example demonstrates a reusable asynchronous algorithm that attaches a transformation function as an intermediate continuation.

Asio/Net.TS

```
template <class F, class Signature>
struct _then_sig;

template <class F, class R, class... Args>
struct _then_sig<F, R(Args...)>
  : std::type_identity<void(std::invoke_result_t<F, Args...>)> {};

template <class F, class Handler>
struct _then_handler {
  F f;
  Handler h;

  void operator()(auto&&... args) {
    std::move(h)(std::move(f)(std::forward<decltype(args)>(args)...));
  }
};

template <template <class, class> class A, class F, class Handler, class C>
struct associator<A, _then_handler<F, Handler>, C> : A<Handler, C> {
  static typename A<Handler, C>::type get(
      const _then_handler<F, Handler>& t, const C& c = C()) noexcept {
    return A<Handler, C>::get(t.h, c);
  }
};

struct _then_init {
  void operator()(auto h, auto f, auto&& init, auto&&... args) {
    std::forward<decltype(init)>(init)(
        _then_handler<decltype(f), decltype(h)>{std::move(f), std::move(h)},
        std::forward<decltype(args)>(args)...);
  }
};

template <class F, class CompletionToken>
struct _then {
  F& f;
```

P2300

```
/// For emulating "deducing this"
template <class A, class B>
  concept __this = same_as<remove_cvref_t<A>, B>;

template<receiver R, class F>
struct _then_receiver {
  R r_;
  F f_;

  // Customize set_value by invoking the callable and passing the result to the inner receiver
  template<class... As>
    requires receiver_of<R, invoke_result_t<F, As...>>
  friend void tag_invoke(std::execution::set_value_t, _then_receiver&& self, As&&... as) {
    std::execution::set_value((R&&) self.r_, invoke((F&&) self.f_, (As&&) as...));
  }

  // Forward all other tag_invoke-based CPOs (copy_cvref_t from P1450):
  template <__this<_then_receiver> Self, class... As, invocable<copy_cvref_t<Self, R>, As...>
Tag>
  friend auto tag_invoke(Tag tag, Self&& self, As&&... as)
    noexcept(is_nothrow_invocable_v<Tag, copy_cvref_t<Self, R>, As...>)
    -> invoke_result_t<Tag, copy_cvref_t<Self, R>, As...> {
    return ((Tag&&) tag)(((Self&&) self).r_, (As&&) as...);
  }
};

template<sender S, class F>
struct _then_sender : std::execution::sender_base {
  S s_;
  F f_;

  template<receiver R>
    requires sender_to<S, _then_receiver<R, F>>
  friend auto tag_invoke(std::experimental::connect_t, _then_sender&& self, R r)
    -> std::execution::connect_result_t<S, _then_receiver<R, F>> {
      return std::execution::connect((S&&) s_, _then_receiver<R, F>{(R&&) r, (F&&) f_});
```

```
  CompletionToken& token;
};

template <class F, class CompletionToken, class... Sigs>
struct async_result<_then<F, CompletionToken>, Sigs...> {
  static auto initiate(auto&& init, _then<F, CompletionToken> x, auto&&... args)
{
    return async_initiate<CompletionToken, typename _then_sig<F, Sigs>::type...>(
        _then_init{}, x.token, x.f, std::forward<decltype(init)>(init),
        std::forward<decltype(args)>(args)...);
  }
};

template <class Op, class F, class CompletionToken>
auto then(Op&& op, F&& f, CompletionToken&& token)
  -> decltype(std::forward<Op>(op)(_then<F, CompletionToken>{f, token})) {
  return std::forward<Op>(op)(_then<F, CompletionToken>{f, token});
}
```

```
  }
};

template<sender S, class F>
sender auto then(S s, F f) {
  return _then_sender{{}, (S&&) s, (F&&) f};
}
```

Usage example: detached

```
then(
    async_write(sock, buf, deferred),
    [](std::error_code e, std::size_t n)
    {
      std::cout << e << "," << n << "\n";
    },
    detached);
```

No additional allocations beyond those used in the wrapped asynchronous operation (which may be zero).

```
start_detached(
    then(
      async_write(sock, buf),
      [](std::error_code e, std::size_t n)
      {
        std::cout << e << "," << n << "\n";
      }));
```

The use of `start_detached` may require an additional allocation beyond those used by the wrapped sender, due to the lifetime requirements of the operation state.

Usage example: eager detached

```
then(
    [&]<class Token>(Token&& token)
    {
      return async_write(sock, buf, std::forward<Token>(token));
    },
    [](std::error_code e, std::size_t n)
```

No equivalent capability.

`` { std::cout << e << "," << n << "\n"; }, detached); ``` Avoids making additional copies of the arguments to the asynchronous operation, which may be expensive. For example, the buffer may contain an atomically reference counted object.	
Usage example: fiber	
`` then(async_write(sock, buf, deferred), [](std::error_code e, std::size_t n) { std::cout << e << "," << n << "\n"; }, use_fiber); ```	``` fiber_wait(then(async_write(sock, buf), [](std::error_code e, std::size_t n) { std::cout << e << "," << n << "\n"; })); ```
Usage example: fiber with efficient consumption of input arguments	
`` then([&]<class Token>(Token&& token) { return async_write(sock, buf, std::forward<Token>(token)); }, [](std::error_code e, std::size_t n) { std::cout << e << "," << n << "\n"; }, use_fiber); ``` Avoids making additional copies of the arguments to the asynchronous operation, which may be expensive. For example, the buffer may contain an atomically reference counted object.	No equivalent capability.
Other discussion	
All asynchronous operations must specify completion signatures. Thus this operation is equivalent to a typed sender.	The `then` algorithm shown here is an untyped sender, which inhibits further composition.

This is a single algorithm that supports eager and lazy uses, in addition to hybrid patterns such as fibers (operation may be launched lazily, but no copy of the arguments is required due to the synchronous completion semantics). The user can choose the semantics that are appropriate to their use case by supplying an appropriate completion token.	In previous revisions of P2300, sender adapters could be: ● lazy, in which case the sender must store a copy of the input arguments, or ● eager, in which case synchronisation is required to ensure that the receiver operations are not invoked until after start is called. Which approach is used was left to the implementer of individual algorithms. In P2300R2 this has changed to mandate strictly lazy submission. Either way, the user is not given a choice and eager use cases are disenfranchised.

Example algorithm: retry

This example demonstrates a reusable asynchronous algorithm that automatically repeats an operation on error.

Asio/Net.TS

```
template <class T> struct is_disposition : std::false_type {};
template <class T> concept disposition = is_disposition<T>::value;

template <> struct is_disposition<std::error_code> : std::true_type {};

auto as_cancelled(std::error_code) {
  return make_error_code(std::errc::operation_canceled);
}

template <> struct is_disposition<std::exception_ptr> : std::true_type {};

auto as_cancelled(std::exception_ptr) {
  return std::make_exception_ptr(std::system_error(
        make_error_code(std::errc::operation_canceled)));
}

template <class Op, class Handler>
struct _retry_handler {
  Op o;
  Handler h;
  cancellation_state c{get_associated_cancellation_slot(h)};

  using cancellation_slot_type = cancellation_slot;

  cancellation_slot_type get_cancellation_slot() const noexcept {
    return c.slot();
```

P2300

```
// _conv needed so we can emplace construct non-movable types into
// a std::optional.
template<invocable F>
  requires std::is_nothrow_move_constructible_v<F>
struct _conv {
  F f_;
  explicit _conv(F f) noexcept : f_((F&&) f) {}
  operator invoke_result_t<F>() && {
    return ((F&&) f_)();
  }
};

// pass through all customizations except set_error, which retries the operation.
template<class O, class R>
struct _retry_receiver {
  O* o_;
  explicit _retry_receiver(O* o) : o_(o) {}
   friend void tag_invoke(std::execution::set_error_t, _retry_receiver&& self, auto&&) noexcept
{
    self.o_->_retry(); // This causes the op to be retried
  }
  // Forward all other tag_invoke-based CPOs (copy_cvref_t from P1450):
  template <__this<_retry_receiver> Self, class... As, invocable<copy_cvref_t<Self, R>, As...>
Tag>
  friend auto tag_invoke(Tag tag, Self&& self, As&&... as)
    noexcept(is_nothrow_invocable_v<Tag, copy_cvref_t<Self, R>, As...>)
```

```
  }

  void operator()(disposition auto e, auto&&... args) {
    if (!e) {
      // Operation succeeded.
      std::move(h)(e, std::forward<decltype(args)>(args)...);
    }
    else if (!!c.cancelled()) {
      // Operation was cancelled.
      std::move(h)(as_cancelled(e), std::forward<decltype(args)>(args)...);
    }
    else {
      // Some other error, try again.
      Op{o}(std::move(*this));
    }
  }
};

template <template <class, class> class A, class Op, class Handler, class C>
struct associator<A, _retry_handler<Op, Handler>, C> : A<Handler, C> {
  static typename A<Handler, C>::type get(
      const _retry_handler<Op, Handler>& t, const C& c = C()) noexcept {
    return A<Handler, C>::get(t.h, c);
  }
};

struct _retry_init {
  void operator()(auto h, auto o) {
    o(_retry_handler<decltype(o), decltype(h)>{o, std::move(h)});
  }
};

template <class CompletionToken>
struct _retry {
  CompletionToken& token;
};

template <class CompletionToken, class... Sigs>
struct async_result<_retry<CompletionToken>, Sigs...> {
  static auto initiate(auto&& init, _retry<CompletionToken> x, auto&&... args) {
    return async_initiate<CompletionToken, Sigs...>(_retry_init{}, x.token,
      [init, args...](auto h) mutable {
        std::move(init)(std::move(h), std::move(args)...);
    });
  }
};

template <class Op, class CompletionToken>
auto retry(Op op, CompletionToken&& token)
  -> decltype(op(_retry<CompletionToken>{token})) {
    -> invoke_result_t<Tag, copy_cvref_t<Self, R>, As...> {
    return ((Tag&&) tag)(((copy_cvref_t<Self, R>&&) self.o_->r_, (As&&) as...);
  }
};

template<sender S>
struct _retry_sender : std::execution::sender_base {
  S s_;
  explicit _retry_sender(S s) : s_((S&&) s) {}

  // Hold the nested operation state in an optional so we can
  // re-construct and re-start it if the operation fails.
  template<receiver R>
  struct _op {
    S s_;
    R r_;
    std::optional<state_t<S&, _retry_receiver<_op, R>>> o_;

    _op(S s, R r): s_((S&&)s), r_((R&&)r), o_{_connect()} {}
    _op(_op&&) = delete;

    auto _connect() noexcept {
      return _conv{[this] {
        return std::execution::connect(s_, _retry_receiver<_op, R>{this});
      }};
    }
    void _retry() noexcept try {
      o_.emplace(_connect()); // potentially throwing
      std::execution::start(*o_);
    } catch(...) {
      std::execution::set_error((R&&) r_, std::current_exception());
    }
    friend void tag_invoke(std::execution::start_t, _op& o) noexcept {
      std::execution::start(*o.o_);
    }
  };

  template<receiver R>
    requires sender_to<S&, _retry_receiver<_op<R>, R>>
  friend _op<R> tag_invoke(std::execution::connect_t, _retry_sender&& self, R r) {
    return _op<R>{(S&&) self.s_, (R&&) r};
  }
};

template<sender S>
sender auto retry(S s) {
  return _retry_sender{(S&&) s};
}
```

```
  return op(_retry<CompletionToken>{token});
}
```

Usage example: callback

```
retry(
    sock.async_connect(endpoint, deferred),
    [](std::error_code e)
    {
      std::cout << e << "," << n << "\n";
    });
```

No additional allocations beyond those used in the wrapped asynchronous operations (which may be zero).

```
start_detached(
    retry(
      then(
        async_connect(sock, endpoint),
        []()
        {
          // ...
        }));
```

The use of `start_detached` may require an additional allocation beyond those used by the wrapped senders, due to the lifetime requirements of the operation state.

Usage example: fiber

```
retry(
    sock.async_connect(endpoint, deferred),
    use_fiber);
```

No additional allocations beyond those used in the wrapped asynchronous operations (which may be zero).

```
fiber_wait(
    retry(
      async_connect(sock, endpoint)));
```

The use of `start_detached` may require an additional allocation beyond those used by the wrapped senders, due to the lifetime requirements of the operation state.

Usage example: fiber with efficient consumption of input arguments

```
retry(
    [&]<class Token>(auto&& token)
    {
      return sock.async_connect(endpoint,
          std::forward<Token>(token));
    },
    use_fiber);
```

Avoids making any copies of the arguments to the retried asynchronous operation, which may be expensive. For a repeated

No equivalent capability. While it is possible to use `defer` to delay construction of the wrapped sender, the copy when constructing the initial sender is unavoidable.

operation such as this, this is safe in the context of fibers, because we know that the operation will complete synchronously.	
Other discussion	
This retry algorithm supports cancellation, and will behave correctly even when supplied with an operation that always fails immediately.	This retry algorithm does not support cancellation correctly, as it relies on the wrapped sender to: ● report cancellation via the `set_done` channel, and ● ensure that it tests for a cancellation request before reporting immediate completion. Neither of these behaviours are mandated in the sender concept.
Immediate completion (on error or otherwise) of an operation never results in a reentrant call to the retry operation.	Immediate error completion may result in a reentrant call to set_error. Consequently this algorithm is susceptible to stack overflow. The user must explicitly break the recursion by adding a sender into the flow that is known to be non-reentrant.
Even when the operation is cancelled, results are propagated to the caller so that they can determine whether any partial side effects were established.	Information about partial side effects is discarded on cancellation.
This retry algorithm works with operations that report errors along with partial side effects.	This retry algorithm is not compatible with senders that report errors along with side effects, as that is only supported on the `set_value` channel.

Addressing other claims in P2464

P2464 says:

> *[...] but if we standardize an executor abstraction, that's what programmers will use, beyond their limited use in the Networking model and its APIs.*
>
> *C++ programmers will write *thousands* of these executors. Yes, you read that correctly. The users of the Networking TS have maybe dozens of them, on a good day.*

This appears to reflect a misaligned mental model about the purpose of the executor and its role in the Asio/Net.TS model, which is to act as a customisation point for the asynchronous "tail calls", as explained above.

On the evidence gathered through many years of using Asio, and one year of P0443 executors in Asio, the vast majority of users never interact with executors directly or, at most, pass existing executor types as policy objects for their asynchronous operations. In certain specialised cases, users may need to write their own bespoke executor implementations. One example of this is to integrate with a GUI event loop.

We suspect that many of these use cases are best addressed by representing the functionality as, in Asio/Net.TS terminology, I/O-objects. This involves exposing the functionality as asynchronous operations. The equivalent of a "scheduler" would be implemented in this way. The GPU example above illustrates this.

P2464 says:

> *The proponents of the Networking TS persistently suggest that this is not a problem. They either suggest that programmers can just add error handling support to a refined executor, or that programmers can just compose programs out of completion handlers and the executor is an insignificant part of this bigger picture.*

As discussed above, executors act as a customisation point for asynchronous "tail calls". This limits the required interface to submission of the continuation. We simply adapt our potentially-failing interface to the "tail call" customisation point requirements.

However, if the intention is to compose an entity with expected-failure semantics into an asynchronous chain, as above we recommend that this be represented as an I/O object.

P2464 says:

> *How many different implementations of this function do I need to support 200 different execution strategies and 10,500 different continuation-decorations(*)?*

> *The 'solutions' suggested by the proponents of the Networking TS model seem to say the answer is "many". With Senders and Receivers, the answer is "one, you just wrote it."*

The answer for the Asio/Net.TS model is: “one”. Since the Asio/Net.TS model is a superset of the capabilities of P2300, we can apply the same techniques. This is illustrated in the examples above.

Conversely, the proposed solution in P2300 forces a single composition mechanism, one for which we have limited field experience, on every user. The alternative of allowing the user to choose between multiple, well-proven composition mechanisms (which may be favoured or required for their domain) seems to be considered a disadvantage.

P2464 says:

> *In the Networking TS, the incoming execution strategy would be an executor. That doesn't work, that doesn't scale, that doesn't integrate to the more-generic, which is also higher-abstraction-level and more-complex. It's not just a simple thread-pool that never fails to run the continuation, because the execution strategy is arbitrarily complex and so is the wrapping continuation.*

This again appears to reflect a misaligned mental model about the purpose of the executor and its role in the Asio/Net.TS model, which is to act as a customisation point for the asynchronous “tail calls”, as explained above.

In the networking context, the actual work of networking is done by sockets, with reading and writing being the main tasks here. Similarly, if the work to be performed is something more complex with possible-failure semantics, like GPUs, the correct approach would be to turn this into an I/O-object, with functionality presented as asynchronous operations.

In addition to fitting with the Asio/Net.TS model, this gives the user control of scheduling and managing the work using all the tools available for coordinating asynchronous operations. See the GPU example above for an illustration of how this may work.

P2464 says:

> *A P0443 executor can't work with [generic algorithms]. It always requires ignoring the whole pseudo-concept of an executor, and getting at a concrete implementation to fetch the error information, and turning the payload data into something that's composable. The whole concept is completely useless, there's no composed code you can write with it.*
>
> *[...]*
>
> *Composing P0443 executors is an ad-hoc exercise. [...] Such a uniform model could perhaps be built on the completion handlers, but it can't be built on executors, and the problem with that is that everybody will build their own incompatible composition model.*

Clearly, we can and do write compositions and generic algorithms on top of the Asio/Net.TS model. In fact, for most compositions, ignoring the concept of an executor is precisely what you should do! The point of executors acting as a customisation point for asynchronous “tail calls” is that they can be treated as a cross-cutting concern to the composition itself.

P2464 says:

> *It has been said that Networking TS needs a guarantee that once an execution context for a continuation has been established, that context is used for all invocations of the continuation, whether an error or a success.*

There is no requirement that says that the Asio/Net.TS needs such a guarantee. There is no single-context requirement baked into the model.

What is baked in: asynchronous operations are required to run the completion handler according to the rules that the associated executor represents. That is, the executor represents a customisation point in the asynchronous algorithms.

This requirement exists to *enable* use cases and improve usability. More generally, it allows users to write their asynchronous agents (chains) so that they always run with a given invariant in place. Use cases include:

- Always on a given thread / thread pool
- Always non-concurrent with respect to a group of other agents
- Always on a GUI main loop
- Optimising for throughput rather than fairness

What the user specifies here via this customisation point is indeed a domain-specific property, and it is specified at the consumer end of the continuation. If the user wants to, the consumer end specifies an associated executor that upholds the invariant.

The model does not insist on it always being present. If the user does nothing, the associated executor is determined to be where the I/O completes i.e. your completion handler is called inline at the point of completion.

P2464 says:

> *[apparently quoting Asio/Net.TS proponents]: "This is not a problem, ASIO has been shipping this for years, and you can find a similar executor in java.util.concurrent."*

As noted above, there is a large body of existing practice employing the same or similar concept to a customisation point for asynchronous “tail calls”.

Problems in P2300

Partial implementation

P2300 assumes the existence of many algorithms which are clearly labeled as "Not yet implemented" in the libunifex repository. A partial list of those unimplemented algorithms includes `transfer, transfer_just, when_all, start_detached`.

Missing exploration of, and support for, different continuation styles

Absent from P2300 and libunifex is visible experience with adapting the model to different continuation styles. Compare this with Asio/Net.TS which supports a variety of continuation styles:

- *CompletionHandler* (callbacks)
- `std::future` or `boost::future`
- `use_awaitable` (C++20 coroutines)
- `deferred` (lazy execution)
- `detached` (null continuation)
- `yield` (fibers / stackful coroutines)
- `std::packaged_task` (yeah, even that thing...)

These are all well tested and used in production systems at many companies. They demonstrate the fitness of the *completion token* design. New continuation styles can be added without breaking the design or existing program. This includes the addition of a `as_sender` completion token, to enable composition with the P2300 task graph model.

Limited exploration of the flexibility afforded by allocators

Evidence suggests that P2300 has not fully explored the design space with respect to the placement of per-operation stable-location memory. Asio/Net.TS employs allocators to their full extent to offer both implementers and users of asynchronous maximum flexibility in determining the locality of POSMs. It provides a default allocation strategy optimized for tail calls found in asynchronous chains. Asio/Net.TS is performant by default - which we believe is an essential default for C++. This allocation strategy is well documented. There is no visible analysis of P2300 covering allocation strategies, no user experience regarding optimized allocation strategies, and no study of the pattern of allocation and deallocation for control flows. While Asio/Net.TS has all of these things, and has had them for decades

In particular, the `start_detached` algorithm, one of the methods provided for consuming senders, by necessity allocates storage for the operation state.

ABI risks in library interfaces based on P2300

Sender operations return their operation state as a structure from the `connect` operation. It is the responsibility of the consumer to ensure the operation state remains valid according to the receiver contract.

Algorithms will store internal variables in this operation state object. More complex algorithms introduce the risk of changes (for example to fix bugs), which can result in a change to the size of the returned state.

Allocator-based designs such as that used by Asio/Net.TS allow ABI sensitive libraries to mitigate this risk. For example, they can request allocations sized in multiples of `max_align_t`.

Operation state lifetimes and concerns over the virality of noexcept

The receiver contract requires that the operation state object must be kept alive until:

- the `set_value` operation exits non-exceptionally, or
- the `set_done` or `set_error` operations are called.

A consequence of this is that, if `set_value` in turn starts a new operation, and `set_value` is not `noexcept`, it must necessarily use a separate operation state location for the newly started operation.

We note that the "control flow" sender algorithms in libunifex, such as `repeat_effect_until`, mark their `set_value` operations as `noexcept`, and furthermore require the `connect` operations to be `noexcept`.

P2300 will likely require language changes to work effectively

We note that the proponents of P2300 already propose changes to the language to accommodate flaws in its design:

- `tail call` keyword or "trampolines" to prevent stack overflow discussed here ( )
- Async RAII to cleanup design flaws discussed here (https://github.com/facebookexperimental/libunifex/blob/2547d87fb98a1fe70eccc170a8c91b144648517d/doc/overview.md#async-cleanup)

It would make sense to explore this space fully before considering adoption of a library that depends on such changes and certainly that's not going to be in a timeframe conducive to inclusion in C++23.

What is set_error for?

Nominally `set_error`'s purpose is to send an "error signal" to a receiver but it's unclear what the exact intention is for "error signals": Do they indicate any failure or only failures from the execution context itself?

If the former, then how is an operation intended to generically distinguish scheduling failures from failures of the operation-in-question? In the world of I/O (particularly network I/O) it is common for failures (what could be construed as an "error signal") to be handled within the context of an operation itself. An `async_connect` may be retried (possibly against a different remote endpoint) for example. This concept starts to lose coherence if scheduling errors are mixed into the domain of "error signals". For example, retrying with a different endpoint makes sense if the endpoint is unreachable, but does it equally make sense if the execution context couldn't execute your work because the system is out of memory?

If the latter then the channel seems doomed to be misused. Users almost certainly won't interpret "error" so narrowly.

The model presented by the Asio/Net.TS solves this problem by separating scheduling errors from I/O failures. This allows each to be dealt with where sufficient context is available: if the scheduling facilities have experienced a failure it seems unlikely an individual operation has the tools at its disposal to remediate the issue. Similarly it is unlikely that other operations against the same underlying scheduling facilities will be able to succeed given those facilities have experienced a failure. A reasonable structure might respond by winding the entire component down and allowing the failure to be emitted where context is available to deal with it (which is what the model of the Asio/Net.TS does in practice).

On the other hand the model presented by P2300 seemingly leaves handling this situation up to the authors of individual operations. Rather than the scheduling component being responsible for handling/reporting its own failures each low level component receives a scheduling failure with the implicit expectation being that it will recognize this, attempt no further progress, and somehow notify the highest level that it ought to shut down.

Is `get_completion_scheduler<set_error_t>` implementable or useful?

In an environment of heterogeneous schedulers it's unclear that `get_completion_scheduler<set_error_t>` is implementable. It makes a promise that the "error signal" will be dispatched via a particular scheduler, but if scheduling work (where such "work" could be sending an "error signal") to run thereupon can itself fail (which is one of the motivations P2464 gives for P2300) it's unclear how this promise can be fulfilled.

While reasoning about where and when the "error signal" is processed may not be important to all users it is certainly important to some users particularly when a certain subset of "error signals" (see above) may be handled by continuing asynchronous processing and accessing data which may be unsafe to access from all execution contexts.

Asio/Net.TS deals with this by separating the two classes of errors. Failures originating from the operation itself are dispatched in the appropriate execution context with all the guarantees thereof whereas scheduling failures are handled separately. This has the

consequence that scheduling errors are handled during the non-generic selection of a concrete execution context where maximum information regarding which errors may be emitted and how to handle them is available. This means that operation authors deal with failures from their chosen sub-operations, while those selecting concrete execution contexts may deal with such errors at the point where a reasonable strategy to handle those failures can sensibly be chosen.

Atomic reference counting costs in cancellation

The cancellation approach shown in P2300 utilises atomic operations in order to correctly manage the lifetime of the registered stop callback. It is not clear if this performance penalty is acceptable, but we will note that different approaches to cancellation do not require atomic synchronisation.

Asio/Net.TS has proven design resilience

Over the years, the Asio/Net.TS design has evolved to add support for new functionality, use cases, and composition techniques. Some examples of this include:

- Cancellation slots are opt-in, a natural addition to the open set of completion handlers' associated characteristics, and may be added to Asio/Net.TS later without breaking the design
- Awaitable completion tokens which support C++20 coroutines are opt-in, fit perfectly into the completion token model, and may be added to Asio/Net.TS later without breaking the design
- Custom executors for I/O objects (wording previously discussed and approved for SG4) don't break the design of existing programs written for the pre-change Net.TS
- Associator traits for completion handlers makes associators an open set without breaking the design of Net.TS or even breaking compilation.
- Variadic `async_result` is a natural extension of the completion token mechanism which doesn't break any existing programs, cause compilation errors, or require changes to the design of Net.TS.

It must be stressed that while improvements to Asio/Net.TS **never contemplate changes to the core C++ language**, additions to the core language have **always been accommodated within the design of Net.TS** without leaving behind existing use-cases, because of its stable foundation. Asio/Net.TS accommodates evolution within its universal framework.

Conclusion: standardize existing practice

Currently "The C++ Standard can't access the Internet." This is embarrassing and long overdue. We should continue with the standardisation of the Networking TS for C++23, with its universal asynchronous model built on extensible completion tokens. Then, we can let P2300 and any necessary evolution take as long as needed. In the meantime, Asio/Net.TS can provide a solid foundation on which P2300 may rest.

P2300 is not a complete asynchronous model, but rather a DSL for composing and creating graphs of operations. It does not specify everything that is required for correct, safe-by-default use of asynchronicity. Users instead rely on assumptions about the underlying implementation, which are unspecified, in order to be able to accommodate P2300's present design choices.

Apart from being a proven, resilient, and much needed addition to the C++ Standard, the Networking TS also meets all the requirements listed in the Chair-Theory guidelines:

- Implementation experience: more than 18 years
- Usage experience: many companies and open source projects, across many domains
- Deployment experience: more than 18 years, both standalone and in Boost

The design is based on principles and practice in common use for at least a decade before that. Furthermore, Asio/Net.TS has been proven as the foundation of multiple second order libraries, such as the widely used **Boost.Beast**, which in turn have enabled the development of a third order library ecosystem.

Acknowledgements

The authors would like to thank Robert Leahy for providing extensive material related to P2300 error handling, and Peter Dimov for suggestions and feedback on the content and structure of this paper.