Document Number: P2480r0
Date: 2021-10-07
Project: Programming Language C++
Audience: LEWG, SG1, SG4
Authors: Jamie Allsop
Christopher Kohlhoff
Klemens Morgenstern
Reply-to: Christopher Kohlhoff <chris@kohlhoff.com>

Response to P2471: “NetTS, Asio, and Sender library design comparison” - corrected and expanded

Introduction

We thank the author of P2471 for taking the time to make these comparisons. We feel it is a good basis for illustrating that the concepts in the P2300 model have direct equivalents in the broader Asio/Net.TS model. The direct comparisons between corresponding features also offer us an opportunity to explore the consequences of the design choices in P2300.

For convenience, we will follow the same document structure as P2471r0. Where there are inaccuracies or incomplete information in P2471r0, we will annotate the corresponding tables below. Furthermore, we have added some comparative examples as an aid to understanding the tables.

Notes

P2471r0 says:

> *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).*

This underscores our point in P2469r0 that the Asio/Net.TS model has been able to evolve to incorporate the use cases represented by P2300, and has done so without leaving behind the existing user base and design techniques that have been employed over many years.

Tables

Asynchronous operation design

Table 1: NetTS - (N4771 is missing P1322, P0958, P1943, P2444)

<pre>concept completion_token:
  async_result<
    completion_token,
    signature>
  ::completion_handler_type;
  async_result<
    completion_token,</pre>
	<pre>concept signature:
	  (ErrorsAndValues...) -> void;
	
	concept completion_handler_type:
	  constructible<
	    completion_handler_type,
	    completion_token>;</pre>

<table>
<tr><td><pre>
    signature>
  ::return_type;
</pre></td><td><pre>
  invocable<
    completion_handler_type,
    signature>;

concept result_type:
  constructible<
    result_type,
    completion_handler_type&>;
</pre></td></tr>
</table>

<table>
<tr><th colspan="2">Table 2: Asio - (Asio has P1322, P0958, P1943, P2444)</th></tr>
<tr><td><pre>
concept completion_token:
  async_result<
    completion_token,
    signature...>
  ::initiate(
    initiation,
    completion_token,
    Args...) -> Result;
</pre></td><td><pre>
concept signature:
  (ErrorsAndValues...) -> void;

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

concept completion_handler:
~~ constructible<~~
~~   completion_handler,~~
~~   completion_token>;~~
  invocable<
    completion_handler,
    signature>...;
</pre></td></tr>
</table>

<table>
<tr><th>Table 3: Sender/Receiver - (P2300)</th></tr>
<tr><td><pre>
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;
</pre></td></tr>
</table>

In P2300, the `operation_state` serves two purposes:

- To carry the information necessary to start the operation between the connect and start calls.
- To provide a per-operation stable memory resource (POSM) for the lifetime of the operation.

In the Asio/Net.TS model, these requirements are decoupled. The first purpose is served by the initiation object and the arguments that accompany it. Once the operation is launched, these are no longer required and can be discarded. If the operation is being used eagerly, the arguments may also be forwarded through without copying. Invocation of the initiation function object is approximately equivalent to calling P2300's `connect` and `start` functions.

The second requirement is satisfied in the Asio/Net.TS model by allowing the operation to manage its own POSMs via an associated allocator (if the operation indeed requires any POSMs at all). With P2300, the owner of the operation is required to manage the `operation_state` lifetime according to the receiver contract, even if no POSM is needed by the operation. As we noted in P2469r0, the `operation_state` size is fixed at compile time and may create ABI risks for APIs that expose senders.

Example: defining an asynchronous operation

In this example we will trivially wrap an existing callback-based API, `read_input`.

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

```
template <typename Callback>
void read_input(const std::string& prompt, Callback cb);

template <class CompletionToken>
auto async_read_input(const std::string& prompt, CompletionToken&& token)
{
  async_completion<CompletionToken, void(std::error_code, std::string)> init(token);

  auto w = make_work_guard(
      get_associated_executor(
        init.completion_handler, system_executor{}));

  auto cb = [w, handler = std::move(init.completion_handler)](std::string input) mutable
  {
    dispatch(w.get_executor(),
        [handler = std::move(handler), input = std::move(input)]() mutable
        {
          std::move(handler)(std::error_code{}, std::move(input));
        }
      );
  };

  read_input(prompt, std::move(cb));

  return init.result.get();
}
```

Example: Asio - (Asio has P1322, P0958, P1943, P2444)

```
template <typename Callback>
void read_input(const std::string& prompt, Callback cb);

template <class CompletionToken>
auto async_read_input(const std::string& prompt, CompletionToken&& token)
{
  auto init = [](auto handler, const std::string& prompt)
  {
    auto ex = prefer(
        get_associated_executor(handler, system_executor{}),
        execution::blocking.possibly,
        execution::outstanding_work.tracked
      );

    auto cb = [ex, handler = std::move(handler)](std::string input) mutable
    {
      execution::execute(ex,
          [handler = std::move(handler), input = std::move(input)]() mutable
          {
```

```
            std::move(handler)(std::error_code{}, std::move(input));
          }
        );
    };

    read_input(prompt, std::move(cb));
  };

  return async_initiate<CompletionToken, void(std::error_code, std::string)>(
      init, token, prompt);
}
```

Example: Sender/Receiver - (P2300)

```
template <typename Callback>
void read_input(const std::string& prompt, Callback cb);

template <class Receiver>
class read_input_operation_state
{
public:
  read_input_operation_state(std::string prompt, Receiver r)
    : prompt_(std::move(prompt)), receiver_(std::move(r)) {}

  void start() noexcept
  {
    try
    {
      read_input(prompt_,
          [this](std::string input)
          {
            try
            {
              execution::set_value(std::move(receiver_), std::move(input));
            }
            catch (...)
            {
              execution::set_error(std::move(receiver_), std::current_exception());
            }
          });
    }
    catch (...)
    {
      execution::set_error(std::move(receiver_), std::current_exception());
    }
  }

private:
  std::string prompt_;
  Receiver receiver_;
};

class read_input_sender
{
public:
  template <template<class...> class Tuple, template<class...> class Variant>
    using value_types = Variant<Tuple<std::string>>;
  template <template<class...> class Variant>
    using error_types = Variant<std::exception_ptr>;
  static constexpr bool sends_done = false;
```

```
  read_input_sender(std::string prompt)
    : prompt_(std::move(prompt))
  {
  }

  template <class Receiver>
  read_input_operation_state<Receiver> connect(Receiver r) &&
  {
    return {std::move(prompt_), std::move(r)};
  }

private:
  std::string prompt_;
};

read_input_sender async_read_input(std::string prompt)
{
  return {std::move(prompt)};
}
```

Example: consuming an asynchronous operation

This example utilises the operation we defined above, using a mechanism for consuming the result at the end of an asynchronous operation chain.

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

```
async_read_input("enter input",
    [](std::error_code e, std::string input)
    {
      if (!e)
        std::cout << "you entered " << input << "\n";
    });
```

Example: Asio - (Asio has P1322, P0958, P1943, P2444)

```
async_read_input("enter input",
    [](std::error_code e, std::string input)
    {
      if (!e)
        std::cout << "you entered " << input << "\n";
    });
```

Example: Sender/Receiver - (P2300)

```
template <typename State>
struct my_input_receiver
{
  State* state_;

  void set_value(std::string input) &&
  {
    std::cout << "you entered " << input << "\n";
    delete state_;
  }

  void set_error(std::exception_ptr) && noexcept
  {
```

```
    delete state_;
  }

  void set_done() && noexcept
  {
    delete state_;
  }
};

struct my_input_operation_state
{
  std::decay_t<
    decltype(
      execution::connect(
        async_read_input(std::declval<std::string>()),
        std::declval<my_input_receiver<my_input_operation_state>>())
    )> operation_state_;

  my_input_operation_state(std::string input)
    : operation_state_(
        execution::connect(
          async_read_input(std::move(input)),
          my_input_receiver<my_input_operation_state>{this}))
  {
  }
};

//...

execution::start((new my_input_operation_state("enter input"))->operation_state_);
```

Initiating function design

Table 4: NetTS - (N4771 is missing P1322, P0958, P1943, P2444)

Any function that takes a completion_token, by convention as the last argument, and returns:

```
(..., completion_token)
  -> decltype(async_result<
    completion_token,
    signature>::return_type;
    signature>::result_type(
      async_result<
        completion_token,
        signature>
          ::completion_handler_type(
            completion_token)));
```

Since this is a convention, one could also put the completion token in front of a variadic argument list.

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

Any function that takes a completion_token, by convention as the last argument, and returns the result of: `(..., completion_token) -> decltype(async_result< completion_token, signature...> ::initiate( initiation, completion_token, Args...));` Since this is a convention, one could also put the completion token in front of a variadic argument list.

Table 6: Sender/Receiver - (P2300)
Any function returning a sender `(...) -> sender;`

Example

Please see the previous section for an example of an initiating function.

Algorithm design

We note that this section in P2471r0 adopts an idiosyncratic, narrow definition of "algorithm", and does not encompass asynchronous algorithm design as a general problem. It appears to refer to algorithms that operate in terms of single asynchronous operations, packaged as an object (a sender, in P2300 terminology).

It should also be noted that what P2300 considers algorithms may be categorised differently in Asio/Net.TS, e.g. transformations of the asynchronous results are usually called completion tokens. Algorithms utilizing IO objects are more often referred to as composed operations, such as `async_read`.

In our experience, the majority of asynchronous algorithms are composed operations, and defined in terms of some concept which contains one or more asynchronous operations within it. For example, an AsyncReadStream concept which defines an `async_read_some` operation, can be notionally defined as:

```
class AsyncReadStream {
public:
  template <class Buffers, class CompletionToken>
  auto async_read_some(Buffers buffers, CompletionToken&& token);
};
```

This concept is then utilised in algorithms such as Asio/Net.TS `async_read`:

```
template <class AsyncReadStream, class Buffers, class CompletionToken>
auto async_read(AsyncReadStream& stream, Buffers buffers, CompletionToken&& token);
```

The equivalent idea in P2300 is represented in the Scheduler concept and algorithms that are expressed in terms of Scheduler.

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

~~Unspecified, but without async_initiate the only option I know of is:~~

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

~~Happy to be corrected.~~

An algorithm is simply an asynchronous operation, with an initiating function as specified above. Thus, the model is self-similar across the layers of abstraction.

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 is an example of this~~

~~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;
```
(struck through)

A packaged asynchronous operation is a unary invocable that takes a completion token:

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

Algorithms over packaged asynchronous operations may look like:

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

Table 9: Sender/Receiver - (P2300)

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

The Asio/Net.TS formulation allows for algorithms to be implemented once, and then used in either a lazy or eager way, depending on the needs of the user. This choice between eager and lazy extends to the algorithms over packaged operations. The user can, furthermore, trivially define their packaged operation as a lambda to avoid copying the input arguments. For example:

```
retry(
    [&](auto&& token)
    {
      return sock.async_send(buf, std::forward<decltype(token)>(token));
    },
    [](std:error_code e, std::size_t n)
    {
```

```
    // ...
  });
```

Please see P2469r0 for more extensive comparative examples of this kind of algorithm.

Example: delegating an operation based on a runtime decision

A simple, yet commonly occurring algorithmic pattern is to wrap alternative asynchronous implementations and choose between them at runtime. This is illustrated in the following class:

```
class async_stream_wrapper {
  stream1_type stream1_;
  stream2_type stream2_;
  int which_;
public:
  //...
  auto async_read_some(...) { ... }
};
```

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

```
template <class Buffers, class CompletionToken>
auto async_read_some(Buffers buffers, CompletionToken&& token)
{
  switch (which_)
  {
  case 1:
    return stream1_.async_read_some(buffers, std::forward<CompletionToken>(token));
  case 2:
    return stream2_.async_read_some(buffers, std::forward<CompletionToken>(token));
  default:
    std::terminate();
  }
}
```

Example: Asio - (Asio has P1322, P0958, P1943, P2444)

```
template <class Buffers, class CompletionToken>
auto async_read_some(Buffers buffers, CompletionToken&& token)
{
  return async_initiate<CompletionToken, void(std::error_code, std::size_t)>(
      [](auto handler, async_stream_wrapper* self, auto buffers)
      {
        switch (self->which_)
        {
        case 1:
          self->stream1_.async_read_some(buffers, std::move(handler));
          break;
        case 2:
          self->stream2_.async_read_some(buffers, std::move(handler));
          break;
        default:
          std::terminate();
        }
      }, token, this, buffers);
}
```

Example: Sender/Receiver - (P2300)

```
template <class Buffers, class Receiver>
class read_some_operation_state
{
  using state1_type = std::decay_t<
    decltype(
      execution::connect(
        std::declval<stream1_type&>().async_read_some(std::declval<Buffers>()),
        std::declval<Receiver>())
    )>;

  using state2_type = std::decay_t<
    decltype(
      execution::connect(
        std::declval<stream2_type&>().async_read_some(std::declval<Buffers>()),
        std::declval<Receiver>())
    )>;

public:
  read_some_operation_state(async_stream_wrapper* self, Buffers buffers, Receiver r)
    : which_(self->which_)
  {
    switch (which_)
    {
    case 1:
      new (&state_.state1_) state1_type{execution::connect(
            self->stream1_.async_read_some(buffers), std::move(r))};
      break;
    case 2:
      new (&state_.state2_) state2_type{execution::connect(
            self->stream2_.async_read_some(buffers), std::move(r))};
      break;
    default:
      std::terminate();
    }
  }

  ~read_some_operation_state()
  {
    switch (which_)
    {
    case 1:
      state_.state1_.~state1_type();
      break;
    case 2:
      state_.state2_.~state2_type();
      break;
    default:
      std::terminate();
    }
  }

  void start() noexcept
  {
    switch (which_)
    {
    case 1:
      state_.state1_.start();
      break;
    case 2:
      state_.state2_.start();
      break;
```

```
    default:
      std::terminate();
    }
  }

private:
  union u
  {
    u() {}
    ~u() {}
    char dummy_;
    state1_type state1_;
    state2_type state2_;
  } state_;
  int which_;
};

template <class Buffers>
class read_some_sender
{
public:
  template <template<class...> class Tuple, template<class...> class Variant>
    using value_types = Variant<Tuple<std::error_code, std::size_t>>;
  template <template<class...> class Variant>
    using error_types = Variant<std::exception_ptr>;
  static constexpr bool sends_done = false;

  read_some_sender(async_stream_wrapper* self, Buffers buffers)
    : self_(self), buffers_(buffers) {}

  template <class Receiver>
  read_some_operation_state<Buffers, Receiver> connect(Receiver r) &&
  {
    return {self_, buffers_, std::move(r)};
  }

private:
  async_stream_wrapper* self_;
  Buffers buffers_;
};

template <class Buffers>
auto async_read_some(Buffers buffers)
{
  return read_some_sender<Buffers>(this, buffers);
}
```

Associated values design

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

```
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;`

The P2300 queries here represent essentially equivalent functionality[1] to the Associator pattern used in Asio/Net.TS, except for a key difference: the provision of a default value, and the ability for the specialised associator to inspect and use this default.

Passing a default to these customisation points enables two important uses:

- The ability to accept the default based on its type, or to reject it and return an alternative.
- The ability to apply additional properties or behaviour to the default, but to otherwise accept the behaviour it represents.

For example, if we want our completion handlers to run on the default executor, but with the addition of mutual exclusion semantics (in pseudocode):

```
auto get_associated_executor(myobj, defaultex)
{
  return make_strand(defaultex);
}
```

Or to specify that our code should always run at a specified priority:

```
auto get_associated_executor(myobj, defaultex)
{
  return require(defaultex, priority(42));
}
```

The use of traits, rather than a CPO-based approach, also allows implementations to detect whether the trait is unspecialised (using a similar approach to that used in P2300's `sender_traits`). This enables the implementation to detect whether the user is explicitly selecting the default, or simply not making any choice at all. In Asio, this technique has been applied to enable additional performance optimisations.

[1] Except that `get_scheduler` is not equivalent to Asio/Net.TS `associated_executor`, as the former has no execution guarantees associated with it.

Executor design

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

`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 table above.

Table 15: Sender/Receiver - (P2300)
~~`concept scheduler:`~~ ~~`  schedule(scheduler) -> sender;`~~ No equivalent.

P2300 schedulers do not play an equivalent role to the executor in the Asio/Net.TS model, and there is no other facility in P2300 that plays a similar role. That is, in conjunction with the `associated_executor` customisation point, Asio/Net.TS executors provide concrete guarantees on where completion handlers are run.

P2300 schedulers may invoke the receiver reentrantly. P2300 provides no facilities that may be used to break the reentrancy and recursion. Asynchronous control flows that are specified as a graph of senders/receivers are susceptible to unfairness, starvation, and stack overflow due to unbounded recursion.

To illustrate unfairness and starvation, consider the following asynchronous operation, implemented as pseudocode:

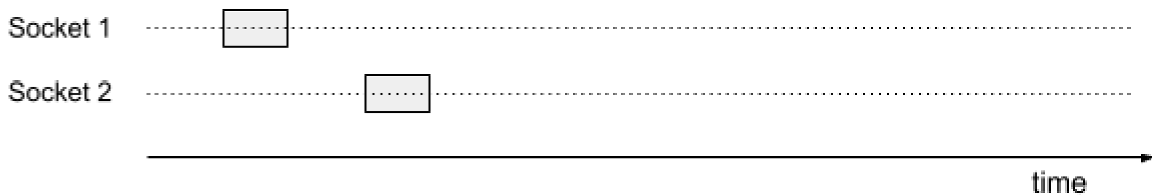
```
void async_read(socket, data, len, callback)
{
  if socket.is_ready_to_read()
  {
    ::read(socket.fd(), data, len);
    callback(len);
  }
  else
    reactor.wait_for_readiness(socket, data, len, callback);
}
```

and application code:

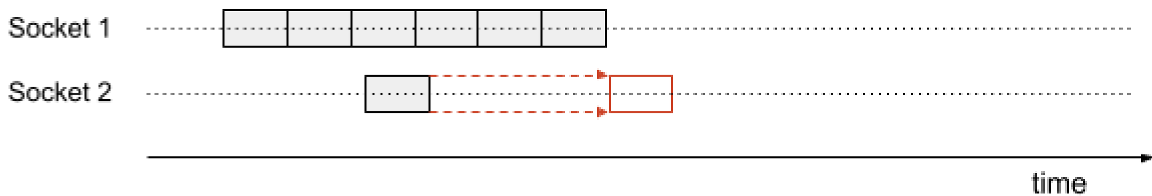
```
void do_read(sock, data, len)
{
  async_read(sock, data, len,
```

```
    [=](auto bytes_transferred)
    {
      // process
      do_read(sock, data, len);
    });
}
```

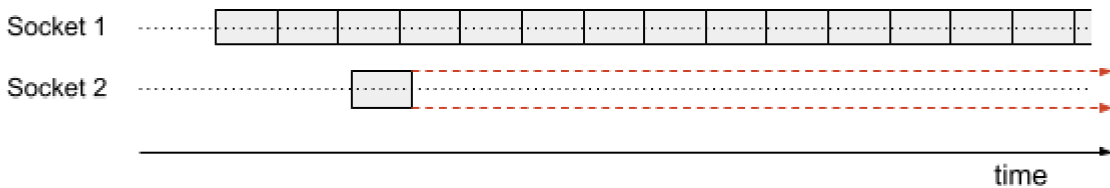
Now suppose our application is single-threaded, handling two connections simultaneously. If data arrives infrequently, clearly there is no issue:



However, if packets arrive for Socket 1 without a break, then with the above pseudocode we do not handle Socket 2's data until all of the Socket 1 messages have been processed. This is unfairness.



Finally, if data arrives for Socket 1 continuously, without a break, Socket 2 will never be processed. This is starvation. Worse, with the above pseudocode, the processing of Socket 1 may occur recursively and ultimately lead to stack overflow.



These are common scenarios in networking, as the arrival rate of events like these is not under the program's control. Indeed, it may even be under the control of a malicious actor that is undertaking a DoS attack on the program.

It may seem tempting to try to fix the problem of stack exhaustion through language changes, such as adding support for tail calls. Unfortunately, this does not address the underlying problems of unfairness and starvation. These choices should not be simply characterised as trade-offs, as they have the potential for costly real-world consequences. We thus remain deeply concerned that the P2300 design thinking is insufficiently paranoid to be suited to internet-facing software development.

While we may indeed want to offer users the ability to prioritise throughput over fairness, many of those who write internet-facing code are unaware of the above dangers, and thus it is important that the library use a safe default. Executors give us this ability to satisfy both of these requirements, and to encode this experience in library form.

Execution context design

Table 16: NetTS - (N4771 is missing 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 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;
  no-requirements
```

Table 18: Sender/Receiver - (P2300)

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